

MODELING THE CRYOSPHERE WITH FEniCS

By

EVAN MICHAEL CUMMINGS

Associates of Applied Art in Audio Production, The Art Institute of Seattle, WA, 2008

Bachelor of Art in Mathematics, University of Montana, Missoula, 2013

Bachelor of Science in Computer Science, University of Montana, Missoula, 2013

Thesis Paper

presented in partial fulfillment of the requirements for the degree of

Master of Science in Computer Science

The University of Montana
Missoula, Montana

August 2016

Approved by:

Scott Whittenburg, Dean of The Graduate School
Graduate School

Douglas Raiford, Chair
Department of Computer Science

Travis Wheeler
Department of Computer Science

Johnathan Bardsley
Department of Mathematics

Modeling the Cryosphere with FEniCS

Chairperson: Douglas Raiford

This manuscript is a collection of problems and solutions related to modeling the cryosphere using the finite element software FEniCS. Included is an introduction to the finite element method; solutions to a variety of problems in one, two, and three dimensions; an overview of popular stabilization techniques for numerically-unstable problems; and an introduction to the governing equations of ice-sheet dynamics with associated FEniCS implementations. The software developed for this project, Cryospheric Problem Solver (CSLVR), is fully open-source and has been designed with the goal of simplifying many common tasks associated with modeling the cryosphere. CSLVR possesses the ability to download popular geological and geographical data, easily convert between geographical projections, develop sophisticated two- or three-dimensional finite-element meshes, convert data between many popular formats, and produce production-quality images of data. Scripts are presented which model the flow of ice using geometry defined by mathematical functions and observed Antarctic and Greenland ice-sheets data. A new way of solving the internal energy distribution of ice to match observed intra-ice water contents within temperate regions is thoroughly explained.



© COPYRIGHT

by

Evan Michael Cummings

2016

All Rights Reserved

To my parents, without whom
this project may never have been
completed.

Acknowledgments

The software newly presented in this manuscript, CSLVR, is largely built from the finite-element software FEniCS (Logg, Mardal, and Wells, 2012). CSLVR solves PDE-constrained-optimization problems through the use of Dolfin-Adjoint (Farrell et al., 2013), which in turn utilizes the IPOPT framework (Wächter and Biegler, 2006) compiled with the Harwell Subroutine Library, a collection of Fortran codes for large scale scientific computation (<http://www.hsl.rl.ac.uk/>). All CSLVR source code is written in the Python programming language (Python Software Foundation, <http://www.python.org>). Countless numerical calculations were accomplished throughout the code-generation process using NumPy and SciPy (Walt, Colbert, and Varoquaux, 2011) via the interactive terminal IPython (Pérez and Granger, 2007). The meshes used for the Greenland and Antarctic simulations were created with GMSH (Geuzaine and Remacle, 2009). All of the code-generated figures were created with Matplotlib (Hunter, 2007). All hand-drawn-vector images were created with the open-source-vector-graphics-software Inkscape. Paraview (Ahrens, Geveci, and Law, 2005) was used extensively to investigate simulation results and generate figures of three-dimensional data. Finally, this manuscript was compiled by the invaluable document-creation software \LaTeX .

Special thanks are given to the following people: my thesis committee for taking the time to provide insight and criticism; Robyn Berg for her thoughtful support throughout my undergraduate and graduate career at the University of Montana; all the faculty in the Departments of Computer Science and Mathematics; The University of Montana Group for Quantitative Study of Snow and Ice for providing many thought-provoking discussions and intuition pertaining to the nature of ice-sheets and glaciers; Douglas Brinkerhoff for creating the software VarGlaS (Brinkerhoff and Johnson, 2013) from which CSLVR evolved; and finally Jesse Johnson for his encouragement, support, and guidance.

Contents

List of Figures	ix
List of Tables	xi
List of Algorithms	xii
I Familiarization with FEniCS	1
1 Basics of finite elements	3
1.1 Motivation: weighted integral approximate solutions	3
1.1.1 Exact solution	4
1.2 The finite element method	4
1.2.1 Variational form	4
1.2.2 Galerkin element equations	5
1.2.3 Local element Galerkin system	6
1.2.4 Globally assembled Galerkin system	6
1.2.5 Imposition of boundary conditions	7
1.2.6 Solving procedure	7
1.2.7 Exact solution	8
1.3 FEniCS framework	8
2 Problems in one dimension	11
2.1 Second-order linear equation	11
2.1.1 Singular perturbation solution	11
2.1.2 Finite element solution	12
2.2 Neumann-Dirichlet problem	12
2.3 Integration	13
2.4 Directional derivative	14
2.5 Eigenvalue problem	15
2.5.1 Fourier series approximation	15
2.5.2 Finite element approximation	16
3 Problems in two dimensions	19
3.1 Poisson equation	19
3.2 Stokes equations with no-slip boundary conditions	20
3.2.1 Stability	21
3.3 Stokes equations with slip-friction boundary conditions	21
4 Problems in three dimensions	27
4.1 Poisson equation	27
4.2 Stokes equations	28
5 Subgrid scale effects	31
5.1 Subgrid scale models	31
5.2 Green's function for \mathcal{L}	32
5.3 Bubbles	32

5.4	Approximation of Green's function for \mathcal{L} with bubbles	33
5.5	Stabilized methods	34
5.6	Diffusion-reaction problem	34
5.6.1	Bubble-enriched solution	34
5.6.2	SSM-stabilized solution	35
5.6.3	Analytic solution	35
5.7	Advection-diffusion-reaction example	36
5.7.1	GLS-stabilized solution	36
5.8	Stabilized Stokes equations	37
6	Nonlinear solution process	41
6.1	Newton-Raphson method	41
6.1.1	Procedure	41
6.1.2	Gâteaux derivatives	42
6.2	Nonlinear problem example	43
6.3	Quasi-Newton solution process	44
7	Optimization with constraints	47
7.1	The control method	47
7.2	Log-barrier solution process	48
II	Dynamics of ice-sheets and glaciers	49
8	Fundamentals of flowing ice	51
8.1	List of symbols	52
9	Momentum and mass balance	55
9.1	Full-Stokes equations	55
9.1.1	Variational principle	55
9.2	First-order approximation	57
9.2.1	Stress tensor simplification	57
9.2.2	Strain tensor simplification	57
9.2.3	First-order vertical velocity and boundary conditions	58
9.2.4	First-order variational principle	58
9.3	Plane-strain approximation	59
9.3.1	Plane-strain variational principle	59
9.4	Reformulated full-Stokes	60
9.4.1	Reformulated-Stokes variational principle	61
9.5	Mass loss due to basal melting	62
9.6	Stokes variational forms	62
9.7	ISMIP-HOM test simulations	63
9.8	Plane-strain simulation	63
10	Internal energy balance	71
10.1	Introduction	71
10.2	Mathematical foundation	72
10.2.1	Momentum interdependence	73
10.3	Energy boundary conditions	74
10.4	Exploring basal-melting-rate	75
10.5	Weak energy approximation	76
10.5.1	Numerical stabilization	76
10.5.2	Energy balance discretization	77
10.6	Water content optimization	78
10.6.1	Variations	79
10.6.2	Energy optimization procedure	79
10.7	Effect of discontinuous energy conductivity	80

11 Thermo-mechanical coupling	83
11.1 Plane-strain simulation	85
12 Inclusion of velocity data	91
12.1 Momentum optimization procedure	91
12.2 Dual optimization for energy and momentum	93
12.3 L-curve analysis	94
12.4 Ice-shelf inversion procedure	96
12.5 ISMIP-HOM inverse test simulation	96
13 Velocity balance	103
13.1 The direction of flowing ice	104
13.1.1 Variational forms for \mathbf{d}	104
13.2 The magnitude of flowing ice	105
13.2.1 Variational form for \bar{u}	105
13.3 Continent-wide simulations	106
13.3.1 Greenland	107
13.3.2 Antarctica	108
14 Stress balance	125
14.1 Membrane stress	125
14.2 Membrane stress balance	126
14.3 ISMIP-HOM test simulation	127
15 Ice age	135
15.1 Variational form	135
16 Application: Jakobshavn	137
16.1 Results	139
16.2 Conclusion	139
A Jump condition at the basal surface	149
B Leibniz formula	151
References	153
Index	156

List of Figures

1.1	FEM intro scratch example	4
1.2	linear Lagrange shape functions	6
1.3	Introductory FEM example solution	8
2.1	Singular-perturbation solution	12
2.2	BVP example one solution	13
2.3	BVP example two solution	14
2.4	Functional derivative example one solution	14
2.5	Functional derivative example two solution	15
2.6	Eigenvector example solution	18
2.7	Comparison of FEM with Fourier series solution	18
3.1	Two-dimension Poisson example	20
3.2	Two-dimensional-no-slip Stokes example	24
3.3	Two-dimensional-slip-friction Stokes example	25
4.1	Three-dimensional Poisson solution	28
4.2	Inside the three-dimensional Poisson solution	28
4.3	Three-dimensional-no-slip Stokes example	30
5.1	The discrete model	31
5.2	One-dimensional bubble function	33
5.3	Two-dimensional bubble function	33
5.4	Diffusion-reaction stabilization example	36
5.5	Advection-diffusion-reaction stabilization example	37
5.6	GLS stabilized slip-friction Stokes example	40
6.1	Newton-Raphson diagram	41
6.2	Nonlinear problem example	44
8.1	Ice-sheet diagram	53
9.1	ISMIP-HOM bedrock topography	63
9.2	ISMIP-HOM momentum experiment velocities	66
9.3	ISMIP-HOM momentum experiment velocity divergence	67
9.4	Three-dimensional ISMIP-HOM momentum solution	68
9.5	Plane-strain momentum experiment solution	69
10.1	Energy diffusivity diagram	73
10.2	Flow-rate-factor diagram	74
10.3	Péclet number and intrinsic-time parameter diagram	77
10.4	Water-content optimization diagram	79
11.1	Plane-strain TMC example basal traction field	86
11.2	Plane-strain water-optimization convergence diagram	88
11.3	Plane-strain TMC convergence diagram	88
11.4	Plane-strain zero-energy-flux solution	89
11.5	Plane-strain water-optimization solution	90

12.5 Inverse ISMIP-HOM ‘true’ data fields	98
12.1 Tikhonov-regularized inverse ISMIP-HOM convergence diagram	99
12.2 ISMIP-HOM Tikhonov L-curve diagram	99
12.3 Total-variation regularized ISMIP-HOM convergence diagram	100
12.4 ISMIP-HOM total-variation L-curve diagram	100
12.6 TV-regularized inverse ISMIP-HOM results	101
12.7 Tikhonov-regularized inverse ISMIP-HOM results with $\gamma_3 = 100$	101
12.8 Tikhonov-regularized inverse ISMIP-HOM results with $\gamma_3 = 500$	102
13.1 Greenland surface-velocity magnitude	110
13.2 Greenland balance-velocity with $\mathbf{d}^{\text{data}} = -\nabla S$	111
13.3 Greenland balance-velocity with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$	112
13.4 Greenland balance-velocity misfit with $\mathbf{d}^{\text{data}} = -\nabla S$	113
13.5 Greenland balance-velocity misfit with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$	114
13.6 Antarctica surface velocity magnitude	115
13.7 Antarctica balance-velocity with $\mathbf{d}^{\text{data}} = -\nabla S$	116
13.8 Antarctica balance-velocity with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$ over shelves.	117
13.9 Antarctica balance-velocity with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$	118
13.10 Antarctica balance-velocity with $\mathbf{d}^{\text{data}} = -\nabla S$ where \mathbf{u}_{ob} are missing.	119
13.11 Antarctica balance-velocity misfit with $\mathbf{d}^{\text{data}} = -\nabla S$	120
13.12 Antarctica balance-velocity misfit with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$ over shelves.	121
13.13 Antarctica balance-velocity misfit with $\mathbf{d}^{\text{data}} = \mathbf{u}_{ob}$	122
13.14 Antarctica balance-velocity misfit with $\mathbf{d}^{\text{data}} = -\nabla S$ where \mathbf{u}_{ob} are missing.	123
14.1 ISMIP-HOM full-Stokes membrane stress	129
14.2 ISMIP-HOM reformulated-Stokes membrane stress	130
14.3 ISMIP-HOM first-order membrane stress	131
14.4 ISMIP-HOM full-Stokes membrane stress balance	132
14.5 ISMIP-HOM reformulated-Stokes membrane stress balance	133
14.6 ISMIP-HOM first-order membrane stress balance	134
16.1 Jakobshavn Glacier mesh	137
16.2 Jakobshavn Glacier topography	138
16.3 Jakobshavn energy profiles	143
16.4 Jakobshavn simulation convergence plot	143
16.5 Jakobshavn simulation results	144
16.6 Effect of water content on basal traction	145
16.7 Jakobshavn membrane stress	146
16.8 Jakobshavn membrane stress balance	147

List of Tables

- 8.1 Empirically-derived-ice-sheet constants 54
- 9.1 ISMIP-HOM momemtum variables 63
- 9.2 Plane-strain momentum variables 64
- 11.1 Plane-strain TMC example variables 86
- 12.1 Inverse ISMIP-HOM variables 97
- 13.1 Greenland balance-velocity variables 107
- 13.2 Antarctica balance-velocity variables 108
- 14.1 Stress-balance ISMIP-HOM variables 128
- 16.1 Jakobshavn simulation variables 139

List of Algorithms

1	Newton-Raphson	42
2	BFGS	45
3	Backtracking line-search	45
4	Thermo-mechanical coupling	84
5	Thermal-parameters update	84
6	Thermo-mechanically coupled data-assimilation	93

Part I

Familiarization with FEniCS

Chapter 1

Basics of finite elements

Many differential equations of interest cannot be solved exactly; however, they may be solved approximately if given some simplifying assumptions. For example, perturbation methods approximate an *inner* and *outer* solution to a problem with different characteristic length or time scales, Taylor-series methods determine a locally convergent approximation, Fourier-series methods determine a globally-convergent approximation, and finite-difference methods provide an approximation over a uniformly discretized domain. The finite element method is a technique which non-uniformly discretizes the domain of a variational or weighted-residual problem into *finite elements*, which may then be assembled into a single matrix equation and solved for an approximate solution.

1.1 Motivation: weighted integral approximate solutions

Following the explanation in Reddy (1993), the approximation of a differential equation with unknown variable u which we seek is given by the linear expansion

$$u(x) \approx \sum_{i=0}^N u_i \psi_i, \quad (1.1)$$

where u_i are coefficients to the solution, N is the number of parameters in the approximation, and ψ is a set of linearly independent functions which satisfy the boundary conditions of the equation. For example, consider the second-order differential equation

$$-\frac{d}{dx} \left[\frac{du}{dx} \right] + u = 0, \quad 0 < x < 1, \quad (1.2)$$

$$u(0) = 1, \quad \left(\frac{du}{dn} \right) \Big|_{x=1} = 0, \quad (1.3)$$

where n is the outward-pointing normal to the domain. In the 1D case here, $n(0) = -1$ and $n(1) = 1$. The $N = 2$ parameter approximation with

$$\psi_0 = 1, \quad \psi_1 = x^2 - 2x, \quad \text{and} \quad \psi_2 = x^3 - 3x,$$

in (1.1) gives the approximate solution

$$u(x) \approx U_N = u_0 + u_1(x^2 - 2x) + u_2(x^3 - 3x).$$

This approximation satisfies the *Neumann* or *natural* boundary condition at $x = 1$, and in order to satisfy the *Dirichlet* or *essential* boundary condition at $x = 0$, we make $u_0 = 1$, producing

$$u(x) \approx U_N = 1 + u_1(x^2 - 2x) + u_2(x^3 - 3x). \quad (1.4)$$

Substituting this approximation into differential equation (1.2) results in

$$\begin{aligned} -2u_1(x-1) - 3u_2(x^2-1) + 1 + u_1(x^2-2x) + u_2(x^3-3x) &= 0 \\ (2u_1 + 3u_2 + 1) - (2u_1 + 2u_1 + 3u_2)x - (3u_2 - u_1)x^2 + u_2x^3 &= 0, \end{aligned}$$

implying that

$$\begin{aligned} 2u_1 + 3u_2 + 1 &= 0 \\ 4u_1 + 3u_2 &= 0 \\ 3u_2 - u_1 &= 0 \\ u_2 &= 0. \end{aligned}$$

This system of equations has only the trivial solution $u = 0$ and is hence inconsistent with differential equation (1.2, 1.3). However, if the problem is evaluated as a *weighted integral* it can be guaranteed that the number of parameters equal the number of linearly independent equations. This weighted integral relation is

$$\int_0^1 w R dx = 0,$$

where R is the approximation residual of Equation (1.2),

$$R = -\frac{d^2 U_N}{dx^2} + U_N,$$

and w are a set of N linearly independent *weight functions*. For this example we use

$$w_1 = x, \quad \text{and} \quad w_2 = x^2,$$

and two integral relations to evaluate,

$$\begin{aligned}
 0 &= \int_0^1 w_1 R dx = \int_0^1 x R dx \\
 &= \left[\frac{1}{2}(2u_1 + 3u_2 + 1)x^2 - \frac{1}{3}(4u_1 + 3u_2)x^3 - \frac{1}{4}(3u_2 - u_1)x^4 + \frac{1}{5}u_2x^5 \right]_0^1 \\
 &= \frac{1}{2}(2u_1 + 3u_2 + 1) - \frac{1}{3}(4u_1 + 3u_2) - \frac{1}{4}(3u_2 - u_1) + \frac{1}{5}u_2 \\
 &= \frac{1}{2} + \left(1 - \frac{4}{3} + \frac{1}{4}\right)u_1 + \left(\frac{3}{2} - 1 - \frac{3}{4} + \frac{1}{5}\right)u_2 \\
 &= \frac{1}{2} - \frac{1}{12}u_1 - \frac{1}{20}u_2, \\
 0 &= \int_0^1 w_2 R dx = \int_0^1 x^2 R dx \\
 &= \left[\frac{1}{3}(2u_1 + 3u_2 + 1)x^3 - \frac{1}{4}(4u_1 + 3u_2)x^4 - \frac{1}{5}(3u_2 - u_1)x^5 + \frac{1}{6}u_2x^6 \right]_0^1 \\
 &= \frac{1}{3}(2u_1 + 3u_2 + 1) - \frac{1}{4}(4u_1 + 3u_2) - \frac{1}{5}(3u_2 - u_1) + \frac{1}{6}u_2 \\
 &= \frac{1}{3} + \left(\frac{2}{3} - 1 + \frac{1}{5}\right)u_1 + \left(1 - \frac{3}{4} - \frac{3}{5} + \frac{1}{6}\right)u_2 \\
 &= \frac{1}{3} - \frac{2}{15}u_1 - \frac{11}{60}u_2,
 \end{aligned}$$

giving a system of equations for the coefficients u_1 and u_2 ,

$$\begin{bmatrix} -\frac{1}{12} & -\frac{1}{20} \\ -\frac{2}{15} & -\frac{11}{60} \end{bmatrix} \cdot \begin{bmatrix} u_1 \\ u_2 \end{bmatrix} = \begin{bmatrix} -\frac{1}{2} \\ -\frac{1}{3} \end{bmatrix}.$$

Solving this system produces $u_1 = \frac{270}{31}$ and $u_2 = \frac{-140}{31}$, and thus approximation (1.4) is given by

$$u_N(x) = 1 + \frac{270}{31}(x^2 - 2x) - \frac{140}{31}(x^3 - 3x). \quad (1.5)$$

1.1.1 Exact solution

Differential equation (1.2) is easily solved exactly:

$$\frac{d^2 u}{dx^2} - u = 0 \implies u = c_1 \cosh(x) + c_2 \sinh(x)$$

$$\begin{aligned}
 u(0) &= c_1 = 1, \quad u'(1) = \sinh(1) + c_2 \cosh(1) = 0 \\
 \implies c_2 &= -\frac{\sinh(1)}{\cosh(1)} = -\tanh(1),
 \end{aligned}$$

and

$$u(x) = \cosh(x) - \tanh(1)\sinh(x). \quad (1.6)$$

Weighted integral approximation (1.5) and exact solution (1.6) are shown in Figure 1.1.

Code Listing 1.1: Scipy source code used to generate Figure 1.1

```

from pylab import *
x = linspace(0,1,1000)
u1 = 270/31.
u2 = -140/31.
u = 1 + u1*(x**2 - 2*x) + u2*(x**3 - 3*x)
e = cosh(x) - tanh(1)*sinh(x)

mpl.rcParams['font.family']      = 'serif'
mpl.rcParams['text.usetex']      = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}']

fig = figure(figsize=(5,3))
ax = fig.add_subplot(111)

ax.plot(x, u, 'k', lw=2.0, label='approximate')
ax.plot(x, e, 'k--', lw=2.0, label='exact')

leg = ax.legend(loc="upper left")
leg.get_frame().set_alpha(0.0)
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$u$')
ax.grid()

tight_layout()
savefig("../images/fenics_intro/weight_int.pdf")

```

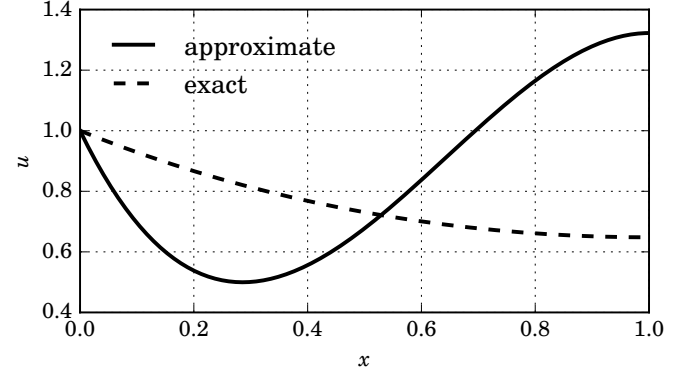


Figure 1.1: Weighted integral approximation (solid) and exact solution (dashed). Note that the approximation will be improved by increasing N .

1.2 The finite element method

The finite element method combines variational calculus, Galerkin approximation methods, and numerical analysis to solve initial and boundary value differential equations (Reddy, 1993). The steps involved in the finite element approximation of a typical problem are

1. Discretization of the domain into finite elements.
2. Derivation of element equations over each element in the mesh.
3. Assembly of local element equations into a global system of equations.
4. Imposition of boundary conditions.
5. Numerical solution of assembled equations.
6. Post processing of results.

In the following sections, we examine each of these steps for the 1D-boundary-value problem (Davis, 2013) over the domain $\Omega \in (0, \ell)$ with essential boundary Γ_D at $x = 0$ and natural boundary Γ_N at $x = \ell$

$$-\frac{d}{dx} \left[k(x) \frac{du}{dx} \right] = f(x) \quad \text{in } \Omega \quad (1.7)$$

$$\left(k(x) \frac{du}{dx} \right) = g_N \quad \text{on } \Gamma_N \quad (1.8)$$

$$u = g_D \quad \text{on } \Gamma_D, \quad (1.9)$$

and develop a finite-element model from scratch.

1.2.1 Variational form

The variational problem corresponding to (1.7 – 1.9) is formed by multiplying Equation (1.7) by the weight function $w(x)$ and integrating over the x -coordinate domain $\Omega \in (0, \ell)$,

$$-\int_{\Omega} \frac{d}{dx} \left[k(x) \frac{du}{dx} \right] w(x) d\Omega = \int_{\Omega} f(x) w(x) d\Omega,$$

with no restrictions on $w(x)$ made thus far. Integrating the left-hand side by parts,

$$\int_0^\ell k \frac{du}{dx} \frac{dw}{dx} dx - \left[wk \frac{du}{dn} \right]_0^\ell = \int_0^\ell f w dx.$$

This formulation is called the *weak form* of the differential equation due to the “weakened” conditions on the approximation of $u(x)$.

In the language of distributional solutions in mathematical analysis, the *trial* or *solution function* u is a member of the *trial* or *solution space* that satisfies the essential boundary condition g_D on Dirichlet boundary Γ_D ,

$$\mathcal{H}_E^1(\Omega) = \{u \in \mathcal{H}^1(\Omega) \mid u = g_D \text{ on } \Gamma_D\}, \quad (1.10)$$

while the *test function* w is member of the *test space*

$$\mathcal{H}_{E_0}^1(\Omega) = \{u \in \mathcal{H}^1(\Omega) \mid u = 0 \text{ on } \Gamma_D\}. \quad (1.11)$$

These spaces are both defined over the space of square-integrable functions whose first derivatives are also square integrable; the $\Omega \subset \mathbb{R}$ *Sobolev space* (Elman, Silvester, and Wathen, 2005)

$$\mathcal{H}^1(\Omega) = \left\{ u : \Omega \rightarrow \mathbb{R} \mid u, \frac{du}{dx} \in L^2(\Omega) \right\}, \quad (1.12)$$

$$L^2(\Omega) = \left\{ u : \Omega \rightarrow \mathbb{R} \mid \int_\Omega u^2 d\Omega < \infty \right\}, \quad (1.13)$$

where the space of functions in $L^2(\Omega)$ is defined with the measure

$$\|u\|_2 = \left(\int_\Omega u^2 d\Omega \right)^{1/2}, \quad (1.14)$$

and the L^2 *inner product* $(f, g) = \int_\Omega f g d\Omega$.

The variational problem consists of finding $u \in \mathcal{H}_E^1(\Omega)$ such that

$$a(u, w) = l(w) \quad \forall w \in \mathcal{H}_E^1(\Omega), \quad (1.15)$$

with *bilinear* term $a(u, w)$ and *linear* term $l(w)$ (Reddy, 1993)

$$a(u, w) = \int_0^\ell k \frac{du}{dx} \frac{dw}{dx} dx - \left[wk \frac{du}{dn} \right]_0^\ell$$

$$l(w) = \int_0^\ell f w dx.$$

The next section demonstrates how to solve the finite-dimensional analog of (1.15) for $U \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ such that

$$\int_0^\ell k \frac{dU}{dx} \frac{d\psi}{dx} dx - \left[\psi k \frac{dU}{dn} \right]_0^\ell = \int_0^\ell f \psi dx \quad (1.16)$$

for all $\psi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$.

1.2.2 Galerkin element equations

Similarly to §1.1, the *Galerkin approximation method* seeks to derive an n -node approximation over a single element e of the form

$$u(x) \approx U^e(x) = \sum_{j=1}^n \psi_j^e(x) u_j^e, \quad (1.17)$$

where u_j^e is the unknown value at node j of element e and ψ^e is a set of n linearly independent approximation functions, otherwise known as *interpolation*, *basis*, or *shape* functions, for each of the n nodes of element e . The approximation functions must be continuous over the element and be differentiable to the same order as the equation.

For the simplest example, the linear interpolation functions with C^0 continuity, known as *Lagrange* interpolation functions defined only over the element interval $x \in [x_i, x_{i+1}]$,

$$\psi_1^e(x) = 1 - \frac{x^e}{h_e} \quad \psi_2^e(x) = \frac{x^e}{h_e}, \quad (1.18)$$

where $x^e = x - x_i$ is the x -coordinate local to element e with first node i and last node $i + 1$, and $h_e = x_{i+1}^e - x_i^e$ is the width of element e (Figure 1.2). Note that these functions are once differentiable as required by the weak form of our example equation, and satisfies the required *interpolation properties*

$$\psi_i^e(x_j^e) = \delta_{ij} \quad \sum_{j=1}^n \psi_j^e(x^e) = 1, \quad (1.19)$$

where δ_{ij} is the *Kronecker delta*,

$$\delta_{ij} = \begin{cases} 0 & \text{if } i \neq j, \\ 1 & \text{if } i = j. \end{cases}$$

The second property in (1.19) implies that the set of functions ψ form a *partition of unity*; this explains how the unknown coefficients u_j^e of approximation (1.17) are equal to the value of approximation u at node j of element e .

Inserting approximation (1.17) into weak form (1.16) integrated over a single element with (not necessarily linear Lagrange) weight functions $w = \psi_i, i = 1, \dots, n$ and add terms for the flux variables interior to the nodes,

$$\int_x k \left(\sum_{j=1}^n u_j^e \frac{d\psi_j^e}{dx} \right) \frac{d\psi_i^e}{dx} dx = \int_x f \psi_i^e dx + \sum_{j=1}^n \psi_i^e(x_j^e) Q_j^e, \quad (1.20)$$

where Q_j^e is the outward flux from node j of element e ,

$$Q_j^e = k \frac{du_j^e}{dn}.$$

Using the second interpolation property in (1.19) the last term in (1.20) is evaluated,

$$\sum_{j=1}^n \psi_i^e(x_j^e) Q_j^e = \sum_{j=1}^n \delta_{ij} Q_j^e = Q_i^e.$$

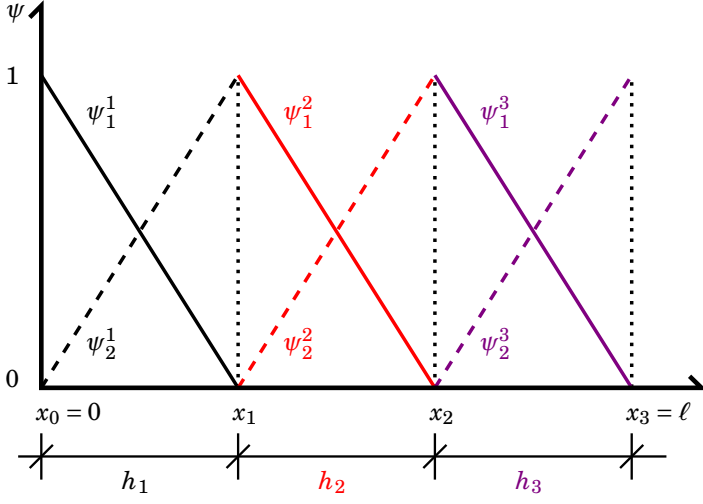


Figure 1.2: Linear Lagrange interpolation functions ψ^e , where the superscript is the element number and subscript the element function number. The even element functions are solid and the odd functions dashed, color-coded by element number.

Next, using the fact that the u_j^e are constant the left-hand-side of (1.20) is re-written

$$\int_x k \left(\sum_{j=1}^n u_j^e \frac{d\psi_j^e}{dx} \right) \frac{d\psi_i^e}{dx} dx = \sum_{j=1}^n u_j^e \int_x k \frac{d\psi_j^e}{dx} \frac{d\psi_i^e}{dx} dx,$$

Therefore, system (1.20) is re-written as

$$\sum_{j=1}^n K_{ij}^e u_j^e = f_i^e + Q_i^e, \quad i = 1, 2, \dots, n,$$

with bilinear and linear terms

$$K_{ij}^e = a(\psi_i^e, \psi_j^e) = \int_x k \frac{d\psi_j^e}{dx} \frac{d\psi_i^e}{dx} dx, \quad f_i^e = l(\psi_i^e) = \int_x f \psi_i^e dx.$$

This is sum is also expressed as the matrix equation

$$K^e \mathbf{u}^e = \mathbf{f}^e + \mathbf{Q}^e. \quad (1.21)$$

Approximations of this kind are referred to as *Galerkin approximations*.

1.2.3 Local element Galerkin system

Using linear Lagrange interpolation functions (1.18) in weak form (1.21) integrated over a single element e of width h_e ,

$$K_{ij}^e = \int_0^{h_e} k_e \frac{d\psi_j^e}{dx} \frac{d\psi_i^e}{dx} dx, \quad f_i^e = \int_0^{h_e} f_e \psi_i^e dx, \quad i, j \in \{1, 2\}.$$

Evaluating the *stiffness matrix* for the element first,

$$\begin{aligned} K_{11}^e &= \int_0^{h_e} k_e \frac{d\psi_1^e}{dx} \frac{d\psi_1^e}{dx} dx & K_{12}^e &= \int_0^{h_e} k_e \frac{d\psi_2^e}{dx} \frac{d\psi_1^e}{dx} dx \\ &= \int_0^{h_e} k_e \left(-\frac{1}{h_e} \right) \left(-\frac{1}{h_e} \right) dx & &= \int_0^{h_e} k_e \left(-\frac{1}{h_e} \right) \left(\frac{1}{h_e} \right) dx \\ &= k_e \left(\frac{x}{h_e^2} \right) \Big|_0^{h_e} & &= -k_e \left(\frac{x}{h_e^2} \right) \Big|_0^{h_e} \\ &= \frac{k_e}{h_e}, & &= -\frac{k_e}{h_e}, \end{aligned}$$

$$\begin{aligned} K_{21}^e &= \int_0^{h_e} k_e \frac{d\psi_1^e}{dx} \frac{d\psi_2^e}{dx} dx & K_{22}^e &= \int_0^{h_e} k_e \frac{d\psi_2^e}{dx} \frac{d\psi_2^e}{dx} dx \\ &= \int_0^{h_e} k_e \left(\frac{1}{h_e} \right) \left(-\frac{1}{h_e} \right) dx & &= \int_0^{h_e} k_e \left(\frac{1}{h_e} \right) \left(\frac{1}{h_e} \right) dx \\ &= -k_e \left(\frac{x}{h_e^2} \right) \Big|_0^{h_e} & &= k_e \left(\frac{x}{h_e^2} \right) \Big|_0^{h_e} \\ &= -\frac{k_e}{h_e}, & &= \frac{k_e}{h_e}, \end{aligned}$$

and the source term f ,

$$\begin{aligned} f_1^e &= \int_0^{h_e} f_e \psi_1^e dx & f_2^e &= \int_0^{h_e} f_e \psi_2^e dx \\ &= \int_0^{h_e} f_e \left(1 - \frac{x}{h_e} \right) dx & &= \int_0^{h_e} f_e \left(\frac{x}{h_e} \right) dx \\ &= f_e \left(x - \frac{x^2}{2h_e} \right) \Big|_0^{h_e} & &= f_e \left(\frac{x^2}{2h_e} \right) \Big|_0^{h_e} \\ &= f_e \left(h_e - \frac{h_e^2}{2h_e} \right) & &= f_e \left(\frac{h_e^2}{2h_e} \right) \\ &= f_e \left(h_e - \frac{h_e}{2} \right) & &= f_e \left(\frac{h_e}{2} \right) \\ &= \frac{1}{2} f_e h_e, & &= \frac{1}{2} f_e h_e. \end{aligned}$$

Finally, the *local element matrix Galerkin system* corresponding to (1.21) with linear-Lagrange elements is

$$\frac{k_e}{h_e} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} u_1^e \\ u_2^e \end{bmatrix} = \frac{f_e h_e}{2} \begin{bmatrix} 1 \\ 1 \end{bmatrix} + \begin{bmatrix} Q_1^e \\ Q_2^e \end{bmatrix}.$$

1.2.4 Globally assembled Galerkin system

In order to connect the set of elements together, extra constraints are imposed on the values interior to the domain. These are

1. The *primary variables* are continuous between nodes such that the last nodal value of an element is equal to its adjacent element's first nodal value,

$$u_n^e = u_1^{e+1}. \quad (1.22)$$

2. The *secondary variables* are balanced between nodes such that outward flux from a connected element is equal to the negative outward flux of its neighboring node,

$$Q_n^e + Q_1^{e+1} = 0. \quad (1.23)$$

If a point source is applied or it is desired to make Q an unknown to be determined,

$$Q_n^e + Q_1^{e+1} = Q_0.$$

First, for global node N ,

$$U_N = u_n^N = u_1^{N+1}, \quad f_N = f_n^N + f_1^{N+1}, \quad \text{and} \quad Q_N = Q_n^N + Q_1^{N+1},$$

and add the last equation from element e to the first equation of element $e+1$,

$$\begin{aligned} \sum_{j=1}^n K_{nj}^e u_j^e + \sum_{j=1}^n K_{1j}^{e+1} u_j^{e+1} &= (f_n^e + Q_n^e) + (f_1^{e+1} + Q_1^{e+1}) \\ \sum_{j=1}^n (K_{nj}^e u_j^e + K_{1j}^{e+1} u_j^{e+1}) &= f_n^e + f_1^{e+1} + Q_n^e + Q_1^{e+1} \\ \sum_{j=1}^n U_j (K_{nj}^e + K_{1j}^{e+1}) &= f_e + Q_e, \end{aligned}$$

which can be transformed into the *global* matrix equation; the Galerkin system

$$K\mathbf{u} = \mathbf{f} + \mathbf{q}, \quad (1.24)$$

where

$$K = \begin{bmatrix} K_{11}^1 & K_{12}^1 & & & & \\ K_{21}^1 & K_{22}^1 + K_{11}^2 & K_{12}^2 & & & \\ & K_{21}^2 & K_{22}^2 + K_{11}^3 & K_{12}^3 & & \\ & & & \ddots & & \\ & & K_{21}^{E-2} & K_{22}^{E-2} + K_{11}^{E-1} & K_{12}^{E-1} & \\ & & & K_{21}^{E-1} & K_{22}^{E-1} + K_{11}^E & K_{12}^E \\ & & & & K_{21}^E & K_{22}^E \end{bmatrix}$$

$$\mathbf{u} = [U_1 \quad U_2 \quad \cdots \quad U_E]^\top$$

$$\mathbf{f} = [f_1 \quad f_2 \quad \cdots \quad f_E]^\top$$

$$\mathbf{q} = [Q_1 \quad Q_2 \quad \cdots \quad Q_E]^\top.$$

Applying Lagrange element equations (1.18), subdividing the domain $x \in [0, \ell]$ into three equal width parts, and making coefficients k and source term f constant throughout the domain, system of equations (1.24) is

$$\frac{k}{h_e} \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} U_1 \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \frac{fh_e}{2} \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} Q_1 \\ Q_2 \\ Q_3 \\ Q_4 \end{bmatrix}, \quad (1.25)$$

a system of four equations and eight unknowns. In the next section, this under-determined system is made solvable by applying boundary conditions and continuity requirements on the internal element flux terms Q_e .

1.2.5 Imposition of boundary conditions

Recall Equation (1.7) is defined with use the essential boundary condition (1.8) and natural boundary condition (1.9). In terms of approximation (10.62), these are respectively

$$u(0) = U_1 = g_D, \quad \left(k \frac{du}{dx} \right) \Big|_{x=\ell} = Q_4 = g_N.$$

Applying continuity requirement for interior nodes (1.23),

$$Q_e = Q_n^e + Q_1^{e+1} = 0, \quad e = 2, 3,$$

to global matrix system (1.25) results in

$$\frac{k}{h_e} \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \cdot \begin{bmatrix} g_D \\ U_2 \\ U_3 \\ U_4 \end{bmatrix} = \frac{fh_e}{2} \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} Q_1 \\ 0 \\ 0 \\ g_N \end{bmatrix}, \quad (1.26)$$

a system of four equations and four unknowns U_2, U_3, U_4 , and Q_1 .

1.2.6 Solving procedure

Before solving global system (1.26), values must be chosen for the known variables and length of the domain. For simplicity, we use

$$g_D = 0, \quad g_N = 0, \quad k = 1, \quad f = 1, \quad h_e = 1/3.$$

With this, system (1.26) simplifies to

$$\begin{bmatrix} 3 & -3 & 0 & 0 \\ -3 & 6 & -3 & 0 \\ 0 & -3 & 6 & -3 \\ 0 & 0 & -3 & 3 \end{bmatrix} \cdot \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} = \frac{1}{6} \begin{bmatrix} 1 \\ 2 \\ 2 \\ 1 \end{bmatrix} + \begin{bmatrix} Q_1 \\ 0 \\ 0 \\ 0 \end{bmatrix}$$

$$\begin{bmatrix} 3 & -3 & 0 & 0 \\ -3 & 6 & -3 & 0 \\ 0 & -3 & 6 & -3 \\ 0 & 0 & -3 & 3 \end{bmatrix} \cdot \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} Q_1 + 1/6 \\ 1/3 \\ 1/3 \\ 1/6 \end{bmatrix}. \quad (1.27)$$

This equation is easily reduced to include only the unknown primary *degrees of freedom* U_e ,

$$\begin{bmatrix} 6 & -3 & 0 \\ -3 & 6 & -3 \\ 0 & -3 & 3 \end{bmatrix} \cdot \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/6 \end{bmatrix}.$$

Because this matrix is square and non-singular, $K = LU$ where L and U are lower- and upper-triangular matrices. Thus the system of equations can be solved by forward and backward substitutions (Watkins, 2010)

$$\begin{aligned} L\mathbf{y} &= \mathbf{q} && \leftarrow \text{forward substitution,} \\ U\mathbf{u} &= \mathbf{y} && \leftarrow \text{backward substitution.} \end{aligned}$$

For stiffness matrix K ,

$$L = \begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 0 & -2/3 & 1 \end{bmatrix} \quad U = \begin{bmatrix} 6 & -3 & 0 \\ 0 & 9/2 & -3 \\ 0 & 0 & 1 \end{bmatrix},$$

and thus

$$Ly = q$$

$$\begin{bmatrix} 1 & 0 & 0 \\ -1/2 & 1 & 0 \\ 0 & -2/3 & 1 \end{bmatrix} \cdot \begin{bmatrix} y_1 \\ y_2 \\ y_3 \end{bmatrix} = \begin{bmatrix} 1/3 \\ 1/3 \\ 1/6 \end{bmatrix}$$

provides $\mathbf{y} = [11/54 \quad 8/27 \quad 2/3]^\top$, which can then be used in backward substitution

$$U\mathbf{u} = \mathbf{y}$$

$$\begin{bmatrix} 6 & -3 & 0 \\ 0 & 9/2 & -3 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} U_2 \\ U_3 \\ U_4 \end{bmatrix} = \begin{bmatrix} 11/54 \\ 8/27 \\ 2/3 \end{bmatrix},$$

producing $\mathbf{u} = [5/18 \quad 4/9 \quad 1/2]^\top$. Finally, Q_1 is solved from the first equation of full system (1.27),

$$\begin{aligned} -3U_2 &= Q_1 + \frac{1}{6} \\ -\frac{15}{18} - \frac{1}{6} &= Q_1 \\ \Rightarrow Q_1 &= -1. \end{aligned}$$

Note that this term is not required to be computed, as the nodal values have been fully discovered. The final three-element solution to this problem is

$$\mathbf{u} = [0 \quad 5/18 \quad 4/9 \quad 1/2]^\top.$$

The flux of quantity u at the left endpoint $x = 0$ is easily calculated:

$$\begin{aligned} Q_1 &= \left(k \frac{du}{dn} \right) \Big|_{x=0} = - \left(\frac{du}{dx} \right) \Big|_{x=0} = -1 \\ \Rightarrow \left(\frac{du}{dx} \right) \Big|_{x=0} &= 1. \end{aligned}$$

1.2.7 Exact solution

The differential equation

$$\begin{aligned} -\frac{d^2 u}{dx^2} &= 1, \quad 0 < x < 1, \\ u(0) &= 0, \quad \frac{du}{dn} \Big|_{x=1} = 0 \end{aligned}$$

is easily solved for the exact solution

$$u_e(x) = -\frac{x^2}{2} + x.$$

The results obtained by hand are compared to the exact solution in Figure 1.3.

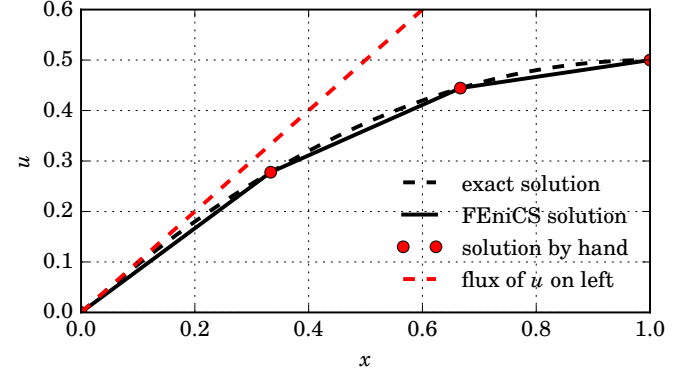


Figure 1.3: Finite element solution computed with FEniCS (solid black), solved exactly (dashed black), and manual finite element (red dots). The slope $Q_1 = 1$ is shown in dashed red.

1.3 FEniCS framework

The FEniCS ([F]inite [E]lement [ni] [C]omputational [S]oftware) package for python and C++ is a set of packages for easily formulating finite element solutions for differential equations (Logg, Mardal, and Wells, 2012). This software includes tools for automatically creating a variety of finite element function spaces, differentiating variational functionals, creating finite element meshes, and much more. It also includes several linear algebra packages for solving the element equations, including PETSc, uBLAS, Epetra, and MTL4.

For example, the finite element code for introductory problem (1.7 – 1.9) is presented in Code Listing 1.2. Notice that only the variational form of the problem is required to find the approximate solution.

Code Listing 1.2: FEniCS source code for the introductory problem.

```
from pylab import *
from fenics import *
from scipy.linalg import lu
from scipy.linalg import solve as sl

# manual solution :
K = array([[6,-3,0],[-3,6,-3],[0,-3,3]])
F = array([1/3., 1/3., 1/6.])

P,L,U = lu(K)
y = sl(L,F)
uf = sl(U,y)
uf = append(0.0, uf)

# FEniCS solution :
mesh = IntervalMesh(3,0,1)
Q = FunctionSpace(mesh, "CG", 1)

f = Constant(1.0)
u = TrialFunction(Q)
v = TestFunction(Q)

a = u.dx(0) * v.dx(0) * dx
l = f * v * dx

def left(x, on_boundary):
    return x[0] == 0 and on_boundary
bc = DirichletBC(Q, 0.0, left)

u = Function(Q)
solve(a == l, u, bc)

# plotting :
xe = linspace(0,1,1000)
xf = linspace(0,1,4)

uv = u.vector().array()[::-1]
ue = -0.5*(xe - 2)*xe
us = xe
```

```

mpl.rcParams['font.family']      = 'serif'
mpl.rcParams['text.usetex']      = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}']

fig = figure(figsize=(5,3))
ax = fig.add_subplot(111)

ax.plot(xe, ue, 'k--', lw=2.0, label='exact solution')
ax.plot(xf, uv, 'k-', lw=2.0, label='FEniCS solution')
ax.plot(xf, uf, 'ro', lw=2.0, label='solution by hand')
ax.plot(xe, us, 'r--', lw=2.0, label='flux of  $u$  on left')

ax.set_xlabel(r' $x$ ')
ax.set_ylabel(r' $u$ ')
ax.set_ylim([0,0.6])
ax.grid()
leg = ax.legend(loc='lower right', fontsize='medium')
leg.get_frame().set_alpha(0.0)
tight_layout()
savefig("../images/fenics_intro/scratch_example.pdf")

```


Chapter 2

Problems in one dimension

To develop an understanding of finite elements further, we investigate several common one-dimensional problems.

2.1 Second-order linear equation

For the first example, we present the inner and outer singular-perturbation solution (Logan, 2006) to the second-order linear equation with non-constant coefficients,

$$\epsilon \ddot{u} - (2t+1)\dot{u} + 2u = 0, \quad 0 < t < 1, \quad 0 < \epsilon \ll 1, \quad (2.1)$$

$$u(0) = 1, \quad u(1) = 0, \quad (2.2)$$

and compare it to the solution obtained using the finite element method.

2.1.1 Singular perturbation solution

The solution to the unperturbed problem ($\epsilon = 0$) is found with the left boundary condition $u(0) = 1$:

$$\begin{aligned} -(2t+1)\dot{u} + 2u &= 0, \\ \implies u(t) &= c_1(2t+1), \end{aligned}$$

applying the boundary condition $u(0) = 1$ results in the outer solution

$$u_o(t) = 2t + 1.$$

In order to determine the width $\delta(\epsilon)$ of the boundary layer we re-scale near $t = 1$ via

$$\xi = \frac{1-t}{\delta(\epsilon)}, \quad U(\xi) = u(t).$$

In scaled variables the differential equation becomes

$$\left(\frac{\epsilon}{\delta(\epsilon)^2}\right)\ddot{U} + \left(\frac{2-2\xi\delta(\epsilon)+1}{\delta(\epsilon)}\right)\dot{U} + 2U = 0$$

For this problem the second derivative term may be retained by making $\delta(\epsilon) = \mathcal{O}(\epsilon)$ resulting in the scaled differential equation

$$\begin{aligned} \left(\frac{\epsilon}{\delta(\epsilon)^2}\right)\ddot{U} + \left(\frac{2-2\xi\delta(\epsilon)+1}{\delta(\epsilon)}\right)\dot{U} + 2U &= 0 \\ \ddot{U} + 3\dot{U} - 2\xi\epsilon\dot{U} + 2\epsilon U &= 0. \end{aligned}$$

The inner approximation to first-order satisfies

$$\ddot{U} + 3\dot{U} = 0,$$

with general solution

$$U(\xi) = C_1 + C_2 e^{-3\xi},$$

and also in terms of u and t ,

$$u(t) = C_1 + C_2 \exp\left(-3\left(\frac{1-t}{\epsilon}\right)\right).$$

Applying the boundary condition $u(1) = 0$ in the boundary layer gives $C_1 = -C_2$, and the inner approximation is

$$u_i(t) = C_2 \left(\exp\left(\frac{3t-3}{\epsilon}\right) - 1 \right).$$

To find C_2 , an overlap domain of order $\sqrt{\epsilon}$ and an appropriate intermediate scaled variable

$$\eta = \frac{1-t}{\sqrt{\epsilon}}.$$

are introduced. Thus $t = 1 - \eta\sqrt{\epsilon}$ and the matching conditions becomes (with η fixed)

$$\lim_{\epsilon \rightarrow 0^+} u_o(1 - \eta\sqrt{\epsilon}) = \lim_{\epsilon \rightarrow 0^+} u_i(1 - \eta\sqrt{\epsilon}),$$

or

$$\begin{aligned} 0 &= + \lim_{\epsilon \rightarrow 0^+} \exp(2(1 - \eta\sqrt{\epsilon}) + 1) \\ &\quad - \lim_{\epsilon \rightarrow 0^+} C_2 \left(\exp\left(\frac{3(1 - \eta\sqrt{\epsilon}) - 3}{\epsilon}\right) - 1 \right) \\ 0 &= + \lim_{\epsilon \rightarrow 0^+} \exp(3 - 2\eta\sqrt{\epsilon}) \\ &\quad - \lim_{\epsilon \rightarrow 0^+} C_2 \left(\exp\left(\frac{-3\eta\sqrt{\epsilon}}{\epsilon}\right) - 1 \right) \\ 0 &= 3 + C_2 \implies C_2 = -3. \end{aligned}$$

A uniform approximation $y_u(t)$ is found by adding the inner and outer approximations and subtracting the common limit in the overlap domain, which is 3 in this case. Consequently,

$$\begin{aligned} u_u(t) &= 2t + 1 - 3 \left(\exp\left(\frac{3t-3}{\epsilon}\right) - 1 \right) - 3 \\ &= 2t - 3 \exp\left(\frac{3t-3}{\epsilon}\right) + 1. \end{aligned}$$

2.1.2 Finite element solution

We arrive at the weak form by taking the inner product of Equation (2.1) with the test function ϕ , integrating over the domain of the problem Ω and integrating the second derivative term by parts,

$$\begin{aligned} 0 &= \int_{\Omega} [\epsilon \ddot{u} - (2t+1)\dot{u} + 2u] \phi d\Omega \\ 0 &= \epsilon \int_{\Omega} \frac{d^2 u}{dt^2} \phi d\Omega - \int_{\Omega} (2t+1) \frac{du}{dt} \phi d\Omega + 2 \int_{\Omega} u \phi d\Omega \\ 0 &= +\epsilon \int_{\Gamma} \left(\frac{du}{dt} \phi \right) n d\Gamma - \epsilon \int_{\Omega} \frac{du}{dt} \frac{d\phi}{dt} d\Omega \\ &\quad - \int_{\Omega} (2t+1) \frac{du}{dt} \phi d\Omega + 2 \int_{\Omega} u \phi d\Omega, \end{aligned}$$

where n is the outward-pointing normal to the boundary Γ . Because the boundary conditions are both Dirichlet, the integral over the boundary are all zero (see test space 1.11). Therefore, the variational problem reads: find $u \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ such that

$$0 = -\epsilon \int_{\Omega} \frac{du}{dt} \frac{d\phi}{dt} d\Omega - \int_{\Omega} (2t+1) \frac{du}{dt} \phi d\Omega + 2 \int_{\Omega} u \phi d\Omega.$$

for all $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$.

A weak solution to this weak problem using linear Lagrange interpolation functions (1.18) is shown in Figure 2.1, and was generated from Code Listing 2.1.

Code Listing 2.1: FEniCS solution to BVP (2.1, 2.2).

```
#####
# finite-element solution :
from fenics import *
mesh = IntervalMesh(1000,0,1)
Q = FunctionSpace(mesh, 'CG', 1)
x = SpatialCoordinate(mesh)[0]
t = mesh.coordinates()[1,0]

v = TestFunction(Q)
u = Function(Q)
du = TrialFunction(Q)

eps = 0.05

r = -eps * u.dx(0) * v.dx(0) * dx \
    - (2*x + 1) * u.dx(0) * v * dx \
    + 2 * u * v * dx

def left(x, on_boundary):
    return on_boundary and x[0] == 0.0

def right(x, on_boundary):
    return on_boundary and x[0] == 1.0

leftBC = DirichletBC(Q, 1.0, left)
rightBC = DirichletBC(Q, 0.0, right)

bcs = [leftBC, rightBC]
J = derivative(r, u, du)

solve(r == 0, u, bcs=bcs, J=J)

uf = u.vector().array()[1:-1]

#####
# singular-perturbation method solution :
from pylab import *

uo = 2*t + 1
ui = -3*(exp((3*t - 3)/eps) - 1)
ul = 3
uu = uo + ui - ul

#####
# plot :
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fourierncn}]

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

ax.set_xlim(0.0, 1.05)
ax.set_ylim(0.0, 3.05)

ax.plot(t, uf, 'k', lw=2.0, label=r"FEniCS")
ax.plot(t, uo, 'r--', lw=2.0, label=r"$u_o(t)$")
```

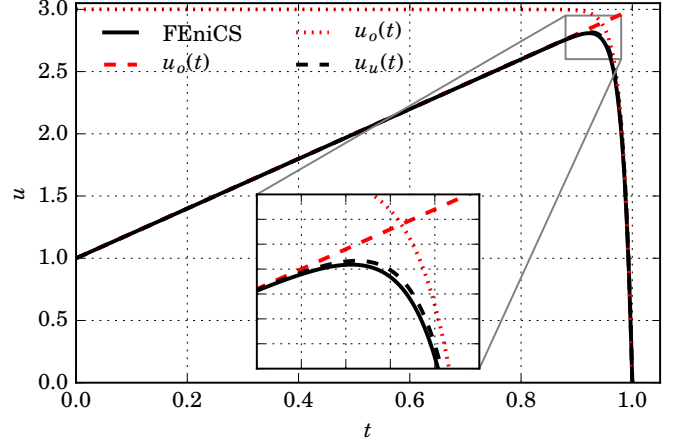


Figure 2.1: Finite element (solid black) and singular perturbation (dashed black) solutions.

```
ax.plot(t, ui, 'r:', lw=2.0, label=r"$u_u(t)$")
ax.plot(t, uu, 'k--', lw=2.0, label=r"$u_u(t)$")

axins = zoomed_inset_axes(ax, 4, loc=8)
axins.set_xlim(0.88, 0.98)
axins.set_ylim(2.60, 2.95)
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
xticks(visible=False)
yticks(visible=False)

axins.plot(t, uf, 'k', lw=2.0, label=r"FEniCS")
axins.plot(t, uo, 'r--', lw=2.0, label=r"$u_o(t)$")
axins.plot(t, ui, 'r:', lw=2.0, label=r"$u_u(t)$")
axins.plot(t, uu, 'k--', lw=2.0, label=r"$u_u(t)$")
axins.grid()

leg = ax.legend(loc='upper left', ncol=2, fontsize='medium')
leg.get_frame().set_alpha(0.0)
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$u$')
ax.grid()

tight_layout()
savefig("../images/fenics_intro/1D_BVP_1.pdf")
```

2.2 Neumann-Dirichlet problem

It may also be of interest to solve a problem possessing a Neumann boundary condition. For example, boundary conditions (2.2) for differential equation (2.1) may be altered to

$$u(0) = 0, \quad \dot{u}(1) = u_r = -10. \quad (2.3)$$

In this case the weak form is derived similarly to §2.1.2, and consists of finding $u \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ such that

$$\begin{aligned} 0 &= +\epsilon \int_{\Gamma} \left(\frac{du}{dt} \phi \right) n d\Gamma - \epsilon \int_{\Omega} \frac{du}{dt} \frac{d\phi}{dt} d\Omega \\ &\quad - \int_{\Omega} (2t+1) \frac{du}{dt} \phi d\Omega + 2 \int_{\Omega} u \phi d\Omega \\ 0 &= +\epsilon \int_{\Gamma_r} u_r \phi d\Gamma_r - \epsilon \int_{\Omega} \frac{du}{dt} \frac{d\phi}{dt} d\Omega \\ &\quad - \int_{\Omega} (2t+1) \frac{du}{dt} \phi d\Omega + 2 \int_{\Omega} u \phi d\Omega, \end{aligned}$$

for all $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$, where the fact that $n = 1$ on the right boundary Γ_r .

The weak solution using linear Lagrange shape functions (1.18) is shown in Figure 2.2, and was generated from Code Listing 2.2.

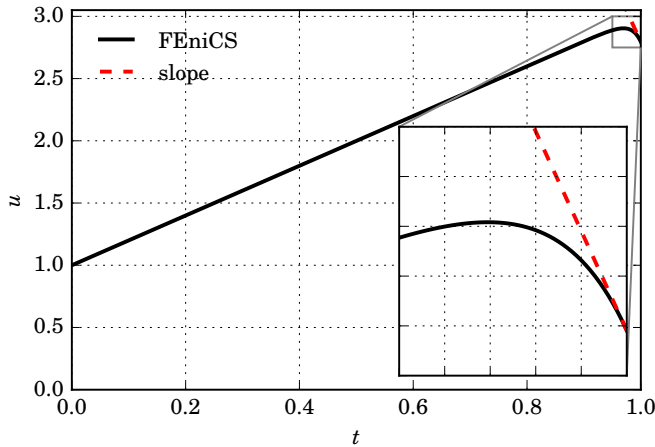


Figure 2.2: Finite element solution (solid black) and slope (dashed red).

Code Listing 2.2: FEniCS solution to BVP (2.1, 2.3).

```
#####
# finite-element solution :
from fenics import *
mesh = IntervalMesh(1000, 0, 1)
Q = FunctionSpace(mesh, 'CG', 1)
x = SpatialCoordinate(mesh)[0]
t = mesh.coordinates()[1, 0]
n = FacetNormal(mesh)

v = TestFunction(Q)
u = Function(Q)
du = TrialFunction(Q)

eps = 0.05
dur = -10.0

r = -eps * u.dx(0) * v.dx(0) * dx \
    + eps * dur * v * ds \
    - (2*x + 1) * u.dx(0) * v * dx \
    + 2 * u * v * dx

def left(x, on_boundary):
    return on_boundary and x[0] == 0.0

def right(x, on_boundary):
    return on_boundary and x[0] == 1.0

leftBC = DirichletBC(Q, 1.0, left)
rightBC = DirichletBC(Q, 0.0, right)

bcs = [leftBC]
J = derivative(r, u, du)

solve(r == 0, u, bcs=bcs, J=J)

uf = u.vector().array()[::-1]

#####
# plot :
from pylab import *
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]

fig = figure(figsize=(5, 3.5))
ax = fig.add_subplot(111)

sf = dur*(t - t[-1]) + uf[-1]

ax.set_xlim(0.0, 1.00)
ax.set_ylim(0.0, 3.05)

ax.plot(t, uf, 'k', lw=2.0, label=r"FEniCS")
ax.plot(t, sf, 'r--', lw=2.0, label=r"slope")

axins = zoomed_inset_axes(ax, 8, loc=4)
axins.set_xlim(0.95, 1.00)
axins.set_ylim(2.75, 3.00)
mark_inset(ax, axins, loc1=2, loc2=4, fc="none", ec="0.5")
xticks(visible=False)
yticks(visible=False)

axins.plot(t, uf, 'k', lw=2.0, label=r"FEniCS")
axins.plot(t, sf, 'r--', lw=2.0, label=r"slope")
axins.grid()

leg = ax.legend(loc='upper left', fontsize='medium')
leg.get_frame().set_alpha(0.0)
ax.set_xlabel(r'$t$')
ax.set_ylabel(r'$u$')
ax.grid()

tight_layout()
savefig("../images/fenics_intro/1D_BVP_2.pdf")
```

2.3 Integration

An interesting problem easily solved with the finite element method is the integration of a function u over domain $\Omega = [a, b]$,

$$v(x) = \int_a^x u(s) ds = U(x) - U(a), \quad (2.4)$$

where U is an anti-derivative of u such that

$$\frac{dU}{dx} = u(x).$$

Because the integral is from a to x , $U(a) = 0$; hence $v(x) = U(x)$ and the equivalent problem to (2.4) is the first-order boundary-value problem

$$\frac{dv}{dx} = u(x), \quad v(a) = 0. \quad (2.5)$$

The corresponding variational problem reads: find $v \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ such that

$$\int_{\Omega} \frac{dv}{dx} \phi d\Omega = \int_{\Omega} u \phi d\Omega,$$

for all $\phi \in S_E^h \subset \mathcal{H}_E^1(\Omega)$.

The linear-Lagrange-element-basis-weak solution to this problem with $u(x) = \cos(x)$ over the domain $\Omega = [0, 2\pi]$ is shown in Figure 2.3, and was generated from Code Listing 2.3.

Code Listing 2.3: FEniCS solution to BVP (2.5)

```
from fenics import *
mesh = IntervalMesh(1000, 0, 2*pi)
Q = FunctionSpace(mesh, 'CG', 1)
x = SpatialCoordinate(mesh)[0]
t = mesh.coordinates()[1, 0]

u = interpolate(Expression('cos(x[0])'), Q)
s = interpolate(Expression('sin(x[0])'), Q)
v = TrialFunction(Q)
phi = TestFunction(Q)

def left(x, on_boundary):
    return on_boundary and abs(x[0]) < 1e-14

# integral is zero on the left
bcs = DirichletBC(Q, 0.0, left)

a = v.dx(0) * phi * dx
L = u * phi * dx
v = Function(Q)
solve(a == L, v, bcs)

uf = u.vector().array()[::-1]
vf = v.vector().array()[::-1]
sf = s.vector().array()[::-1]

r = vf - sf

#####
# plot :
from pylab import *
from mpl_toolkits.axes_grid1.inset_locator import zoomed_inset_axes
from mpl_toolkits.axes_grid1.inset_locator import mark_inset

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]

fig = figure(figsize=(5, 3.5))
ax = fig.add_subplot(111)

ax.plot(t, uf, 'k', lw=2.0, label=r'$u$')
ax.plot(t, vf, 'r--', lw=2.0, label=r'$v$')
ax.plot(t, r, '#880000', lw=2.0, label=r'$\epsilon$')

ax.set_xlim(0, 2*pi)
leg = ax.legend(loc='upper center', fontsize='medium')
leg.get_frame().set_alpha(0.0)
ax.set_xlabel(r'$x$')
ax.grid()

tight_layout()
savefig("../images/fenics_intro/1D_BVP_4.pdf")
```

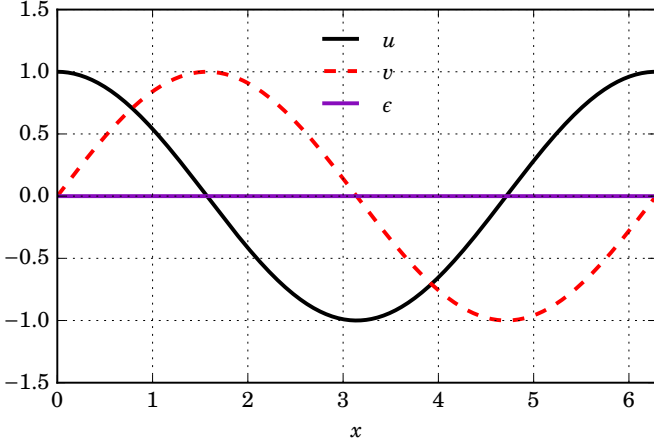


Figure 2.3: The function $u(x) = \cos(x)$ for $a = 0$, $b = 2\pi$ (solid black); finite element solution $v(x) \approx \sin(x)$ (dashed red); and error $\epsilon(x) = v(x) - \sin(x)$ (purple solid).

2.4 Directional derivative

It is often important to compute the derivative of one function with respect to another, say

$$w(x) = \frac{du}{dv} = \frac{du}{dx} \frac{dx}{dv} = \frac{du}{dx} \left(\frac{dv}{dx} \right)^{-1}, \quad (2.6)$$

for continuous functions $u(x)$ and $v(x)$ defined over the interval $\Omega = [a, b]$. The variational form for this problem with trial function $w \in M^h \subset L^2(\Omega)$ (see L^2 space (1.13)), test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$, is simply the inner product

$$\int_{\Omega} w \phi d\Omega = \int_{\Omega} \frac{du}{dx} \left(\frac{dv}{dx} \right)^{-1} \phi d\Omega, \quad \forall \phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega),$$

where no restrictions are made on the boundary; this is possible because both u and v are known *a priori* and hence their derivatives may be computed directly and estimated throughout the entire domain.

Solving problems of this type are referred to in the literature as *projections*, due to the fact that they simply project a known solution onto a finite element basis.

An example solution with $u(x) = \sin(x)$, $v(x) = \cos(x)$ is generated using linear-Lagrange elements from Code Listing 2.4 and depicted in Figure 2.4, and another with $u(x) = 3x^4$, $v(x) = x^6$ generated from Code Listing 2.5 and depicted in Figure 2.5.

Code Listing 2.4: FEniCS solution to BVP (2.6) with $u(x) = \sin(x)$, $v(x) = \cos(x)$.

```
from pylab import *
from fenics import *

xmin = 0
xmax = 2*pi
mesh = IntervalMesh(1000, xmin, xmax)
Q = FunctionSpace(mesh, 'CG', 1)

u = interpolate(Expression('sin(x[0])'), Q)
v = interpolate(Expression('cos(x[0])'), Q)
dudv = interpolate(Expression('-cos(x[0])/sin(x[0])'), Q)
dudv_1 = u.dx(0) * 1/v.dx(0)
```

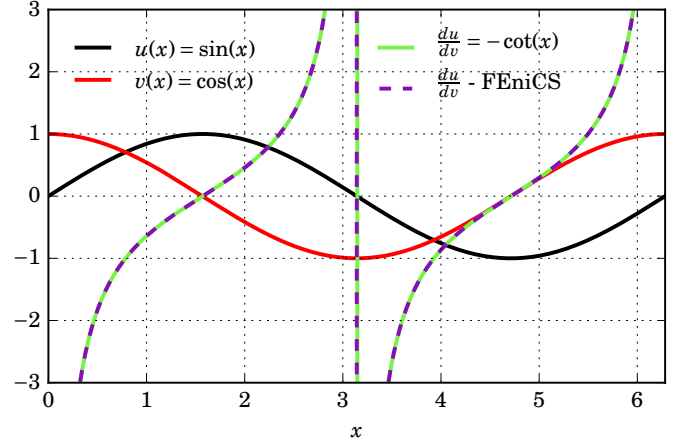


Figure 2.4: Both analytic (green) and numerical (purple) functional derivative $\frac{du}{dv} = -\cot(x)$ for $u(x) = \sin(x)$ (black), and $v(x) = \cos(x)$ (red).

```
x = mesh.coordinates()[0,:][:-1]
u_v = u.vector().array()
v_v = v.vector().array()
d_va = dudv.vector().array()
d_v1 = project(dudv_1).vector().array()

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

purp = '#880cbc'
grun = '#77f343'

ax.plot(x, u_v, 'k', lw=2.0, label=r'$u(x) = \sin(x)$')
ax.plot(x, v_v, 'r', lw=2.0, label=r'$v(x) = \cos(x)$')

ax.plot(x, d_va, color=grun, ls='--', lw=2.0,
        label=r'$\frac{du}{dv} = -\cot(x)$')
ax.plot(x, d_v1, color=purp, ls='--', lw=2.0,
        label=r'$\frac{du}{dv} = \text{FEniCS}$')

ax.grid()
ax.set_xlabel(r'$x$')
ax.set_ylim([-3,3])
ax.set_xlim([0, 6])
leg = ax.legend(loc='upper left', ncol=2, columnspacing=5, fontsize='medium')
leg.get_frame().set_alpha(0.0)

tight_layout()
savefig('.../images/fenics_intro/1D_dir_dir_1.pdf')
```

Code Listing 2.5: FEniCS solution to BVP (2.6) with $u(x) = 3x^4$, $v(x) = x^6$.

```
from pylab import *
from fenics import *

mesh = IntervalMesh(1000, -1, 1)
Q = FunctionSpace(mesh, 'CG', 1)

u = interpolate(Expression('pow(x[0], 6)'), Q)
v = interpolate(Expression('pow(x[0], 2)'), Q)
dudv = interpolate(Expression('3*pow(x[0], 4)'), Q)

dudv_1 = u.dx(0) * 1/v.dx(0)

x = mesh.coordinates()[0,:][:-1]
u_v = u.vector().array()
v_v = v.vector().array()
d_va = dudv.vector().array()
d_v1 = project(dudv_1).vector().array()

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

purp = '#880cbc'
grun = '#77f343'

ax.plot(x, u_v, 'k', lw=2.0, label=r'$u(x)=x^6$')
ax.plot(x, v_v, 'r', lw=2.0, label=r'$v(x)=x^2$')

ax.plot(x, d_va, color=grun, ls='--', lw=2.0,
        label=r'$\frac{du}{dv} = 3x^4$')
ax.plot(x, d_v1, color=purp, ls='--', lw=2.0,
        label=r'$\frac{du}{dv} = \text{FEniCS}$')

ax.grid()
ax.set_xlabel(r'$x$')
leg = ax.legend(loc='upper center', ncol=2, fontsize='medium')
leg.get_frame().set_alpha(0.0)

tight_layout()
```

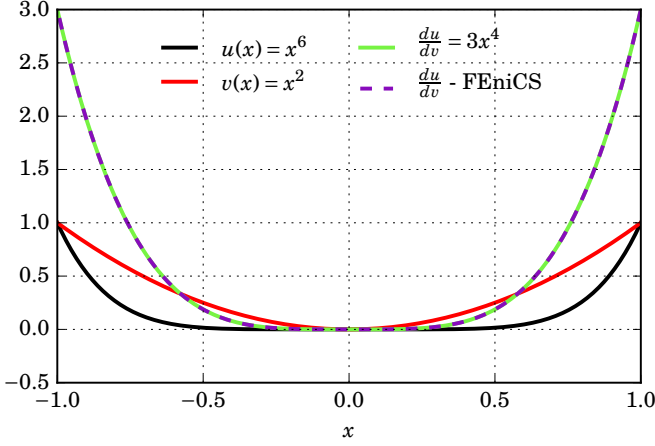



Figure 2.5: Both analytic (green) and numerical (purple) functional derivative $\frac{du}{dv} = 3x^4$ for $u(x) = x^6$ (black), and $v(x) = x^2$ (red).

`savefig('.../images/fenics_intro/1D_dir_dir_2.pdf')`

2.5 Eigenvalue problem

The 1D initial-boundary-value problem for the heat equation with heat conductivity k is

$$\frac{\partial u}{\partial t} = \frac{\partial}{\partial x} \left[k \frac{\partial u}{\partial x} \right], \quad 0 < x < \ell, \quad t > 0, \quad (2.7)$$

$$u(0, t) = u(\ell, t) = 0, \quad t > 0, \quad (2.8)$$

$$u(x, 0) = f(x), \quad 0 < x < \ell. \quad (2.9)$$

We examine this problem in the context of exact and approximate Eigenvalues and Eigenvectors in the following sections.

2.5.1 Fourier series approximation

The solution of system (2.7 – 2.9) can be approximated by the method of separation of variables, developed by Joseph Fourier in 1822 (Davis, 2013). Using this method, we first assume a solution of the form $u(x, t) = X(x)T(t)$, so that equation (2.7) reads

$$X(x) \frac{\partial T(t)}{\partial t} = \frac{\partial}{\partial x} \left[k \frac{\partial X(x)}{\partial x} \right] T(t)$$

$$\frac{1}{kT(t)} \frac{\partial T(t)}{\partial t} = \frac{1}{X(x)} \frac{\partial}{\partial x} \left[\frac{\partial X(x)}{\partial x} \right],$$

For two functions with two different independent variables to be equal, they must both equal to the same constant $-\lambda$, $\lambda > 0$,

$$\frac{1}{kT(t)} \frac{\partial T(t)}{\partial t} = \frac{1}{X(x)} \frac{\partial}{\partial x} \left[\frac{\partial X(x)}{\partial x} \right] = -\lambda,$$

so

$$\frac{\partial T(t)}{\partial t} + \lambda k T(t) = 0, \quad \frac{\partial^2 X(x)}{\partial x^2} + \lambda X(x) = 0. \quad (2.10)$$

The solution to the first-order T equation is

$$T(t) = A e^{-\lambda k t}, \quad (2.11)$$

while the solution to the second-order X equation is

$$X(x) = B_1 \sin(\sqrt{\lambda} x) + B_2 \cos(\sqrt{\lambda} x),$$

for some coefficients A , B_1 , and B_2 where B_1 and B_2 cannot both be zero since then only the trivial solution exists. Applying boundary conditions (2.8), $X(0) = X(\ell) = 0$,

$$X(0) = B_2 = 0,$$

$$X(\ell) = B_1 \sin(\sqrt{\lambda} \ell) = 0$$

$$\Rightarrow \sqrt{\lambda} \ell = n\pi, \quad n = 1, 2, 3, \dots$$

$$\Rightarrow \lambda = \left(\frac{n\pi}{\ell} \right)^2, \quad n = 1, 2, 3, \dots$$

Therefore, the Eigenfunctions and Eigenvalues for the steady-state problem are

$$X_n(x) = \sin(\sqrt{\lambda_n} x) \quad \lambda_n = \left(\frac{n\pi}{\ell} \right)^2, \quad n = 1, 2, 3, \dots \quad (2.12)$$

By the Superposition Principle, any linear combination of solutions is again a solution, and solution to the transient problem is

$$\begin{aligned} u(x, t) &= \sum_{n=1}^{\infty} c_n X_n(x) T_n(t) \\ &= \sum_{n=1}^{\infty} c_n \sin(\sqrt{\lambda_n} x) e^{-\lambda_n k t}, \end{aligned} \quad (2.13)$$

where $c_n = AB_1$ and (2.11) has been used with $\lambda_n = \left(\frac{n\pi}{\ell} \right)^2$. The coefficient c_n may be discovered by inspecting initial condition (2.9)

$$u(x, 0) = \sum_{n=1}^{\infty} c_n \sin(\sqrt{\lambda_n} x) = f(x).$$

Multiplying both sides of this function by $\sin(m\pi x/\ell)$ for arbitrary m and utilizing the fact that

$$\int_0^{\ell} \sin\left(\frac{n\pi x}{\ell}\right) \sin\left(\frac{m\pi x}{\ell}\right) dx = \begin{cases} 0, & n \neq m, \\ \ell/2, & n = m, \end{cases}$$

results in

$$\begin{aligned} \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{\ell}\right) \sin\left(\frac{m\pi x}{\ell}\right) &= f(x) \sin\left(\frac{m\pi x}{\ell}\right) \\ \int_0^{\ell} \sum_{n=1}^{\infty} c_n \sin\left(\frac{n\pi x}{\ell}\right) \sin\left(\frac{m\pi x}{\ell}\right) dx &= \int_0^{\ell} f(x) \sin\left(\frac{m\pi x}{\ell}\right) dx \\ \sum_{n=1}^{\infty} c_n \int_0^{\ell} \sin\left(\frac{n\pi x}{\ell}\right) \sin\left(\frac{m\pi x}{\ell}\right) dx &= \int_0^{\ell} f(x) \sin\left(\frac{m\pi x}{\ell}\right) dx \\ c_m \int_0^{\ell} \sin\left(\frac{m\pi x}{\ell}\right) \sin\left(\frac{m\pi x}{\ell}\right) dx &= \int_0^{\ell} f(x) \sin\left(\frac{m\pi x}{\ell}\right) dx \\ c_m \left(\frac{\ell}{2} \right) &= \int_0^{\ell} f(x) \sin\left(\frac{m\pi x}{\ell}\right) dx. \end{aligned}$$

Finally, replacing the dummy variable m with n ,

$$c_n = \frac{2}{\ell} \int_0^\ell f(x) \sin\left(\frac{n\pi x}{\ell}\right) dx. \quad (2.14)$$

Therefore, the Fourier series approximation to (2.7 – 2.9) is (2.13) with coefficients given by (2.14).

2.5.2 Finite element approximation

Investigating the Eigenvalue problem of separated equations (2.10) for X ,

$$\frac{\partial^2 X(x)}{\partial x^2} + \lambda X(x) = 0, \quad X(0) = X(\ell) = 0,$$

suggests that a weak form can be developed by making $X \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ and taking the inner product of this equation with the test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$

$$\begin{aligned} \int_0^\ell \left[\frac{\partial^2 X}{\partial x^2} + \lambda X \right] \phi dx &= 0 \\ \int_0^\ell \left[\lambda X \phi - \frac{\partial X}{\partial x} \frac{\partial \phi}{\partial x} \right] dx - \frac{\partial X}{\partial x} \phi(0) + \frac{\partial X}{\partial x} \phi(\ell) &= 0 \\ \int_0^\ell \left[\lambda X \phi - \frac{\partial X}{\partial x} \frac{\partial \phi}{\partial x} \right] dx &= 0, \end{aligned} \quad (2.15)$$

where the fact that the boundary conditions are all essential has been used.

Substituting the Galerkin approximation

$$X_e = \sum_{j=1}^n u_j^e \psi_j^e(x),$$

where u_j^e is the j th nodal value of X at element e and $\psi_j^e(x)$ is the element's associated interpolation function, into Equation (2.15) results in the Galerkin system

$$\left[\lambda M_{ij}^e - K_{ij}^e \right] \cdot \mathbf{u}^e = \mathbf{0},$$

with stiffness tensor K and mass tensor M

$$K_{ij}^e = \int_0^\ell \frac{\partial \psi_i^e}{\partial x} \frac{\partial \psi_j^e}{\partial x} dx, \quad M_{ij}^e = \int_0^\ell \psi_i^e \psi_j^e dx.$$

Expanding the element equation tensors as in §1.2.3 results in

$$\left(\frac{\lambda h_e}{6} \begin{bmatrix} 2 & 1 \\ 1 & 2 \end{bmatrix} - \frac{1}{h_e} \begin{bmatrix} 1 & -1 \\ -1 & 1 \end{bmatrix} \right) \cdot \begin{bmatrix} u_1^e \\ u_2^e \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

For a concrete example, we assemble this local system over an equally-space $n = 3$ -element function space, implying that $h_e = 1/3$ and (see §1.2.4)

$$\left(\frac{\lambda}{18} \begin{bmatrix} 2 & 1 & 0 & 0 \\ 1 & 4 & 1 & 0 \\ 0 & 1 & 4 & 1 \\ 0 & 0 & 1 & 2 \end{bmatrix} - 3 \begin{bmatrix} 1 & -1 & 0 & 0 \\ -1 & 2 & -1 & 0 \\ 0 & -1 & 2 & -1 \\ 0 & 0 & -1 & 1 \end{bmatrix} \right) \cdot \begin{bmatrix} X_1 \\ X_2 \\ X_3 \\ X_4 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ 0 \\ 0 \end{bmatrix}.$$

The boundary conditions $X(0) = X(\ell) = 0$ imply that $X_1 = X_4 = 0$ and thus this system reduces to a system of two linear equations,

$$\begin{aligned} \left(\lambda \frac{4}{18} - 6 \right) X_2 + \left(\lambda \frac{1}{18} + 3 \right) X_3 &= 0 \\ \left(\lambda \frac{1}{18} + 3 \right) X_2 + \left(\lambda \frac{4}{18} - 6 \right) X_3 &= 0, \end{aligned}$$

or

$$\begin{bmatrix} \left(\lambda \frac{2}{9} - 6 \right) & \left(\lambda \frac{1}{18} + 3 \right) \\ \left(\lambda \frac{1}{18} + 3 \right) & \left(\lambda \frac{2}{9} - 6 \right) \end{bmatrix} \cdot \begin{bmatrix} X_2 \\ X_3 \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}.$$

The characteristic polynomial is found by setting the determinant of the coefficient matrix equal to zero,

$$\begin{aligned} \left| \begin{bmatrix} \left(\lambda \frac{2}{9} - 6 \right) & \left(\lambda \frac{1}{18} + 3 \right) \\ \left(\lambda \frac{1}{18} + 3 \right) & \left(\lambda \frac{2}{9} - 6 \right) \end{bmatrix} \right| &= 0 \\ \left(\lambda \frac{2}{9} - 6 \right) \left(\lambda \frac{2}{9} - 6 \right) - \left(\lambda \frac{1}{18} + 3 \right) \left(\lambda \frac{1}{18} + 3 \right) &= 0 \\ \left(\lambda^2 \frac{4}{81} - \lambda \frac{24}{9} + 36 \right) - \left(\lambda^2 \frac{1}{324} + \lambda \frac{6}{18} + 9 \right) &= 0 \\ \left(\frac{4}{81} - \frac{1}{324} \right) \lambda^2 - \left(\frac{24}{9} + \frac{6}{18} \right) \lambda + 27 &= 0 \\ \left(\frac{5}{108} \right) \lambda^2 - 3\lambda + 27 &= 0, \end{aligned}$$

with roots

$$\lambda = \frac{3 \pm \sqrt{9 - 108 \left(\frac{5}{108} \right)}}{2 \left(\frac{5}{108} \right)} = \frac{162}{5} \pm \frac{108}{5},$$

providing the two Eigenvalue approximations $\tilde{\lambda}_1 = 54/5 = 10.8$ and $\tilde{\lambda}_2 = 54$. The Eigenvectors may then be derived from the linear systems

$$\begin{aligned} \begin{bmatrix} \left(\tilde{\lambda}_1 \frac{2}{9} - 6 \right) & \left(\tilde{\lambda}_1 \frac{1}{18} + 3 \right) \\ \left(\tilde{\lambda}_1 \frac{1}{18} + 3 \right) & \left(\tilde{\lambda}_1 \frac{2}{9} - 6 \right) \end{bmatrix} \cdot \begin{bmatrix} X_2^1 \\ X_3^1 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} \left(\tilde{\lambda}_2 \frac{2}{9} - 6 \right) & \left(\tilde{\lambda}_2 \frac{1}{18} + 3 \right) \\ \left(\tilde{\lambda}_2 \frac{1}{18} + 3 \right) & \left(\tilde{\lambda}_2 \frac{2}{9} - 6 \right) \end{bmatrix} \cdot \begin{bmatrix} X_2^2 \\ X_3^2 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} -\frac{18}{5} & \frac{18}{5} \\ \frac{18}{5} & -\frac{18}{5} \end{bmatrix} \cdot \begin{bmatrix} X_2^1 \\ X_3^1 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \\ \begin{bmatrix} 6 & 6 \\ 6 & 6 \end{bmatrix} \cdot \begin{bmatrix} X_2^2 \\ X_3^2 \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix} \end{aligned}$$

providing $X^1 = [1 \quad 1]^\top$ and $X^2 = [1 \quad -1]^\top$.

Similar to the derivation of Fourier series solution (2.13), these approximate Eigenvectors and Eigenvalues are used to create the approximate three-element transient solution

$$\begin{aligned} u(x, t) &\approx \tilde{u}(x, t) = \sum_{e=1}^n X^e(x) T^e(t, \tilde{\lambda}_e) \\ &= \sum_{e=1}^3 \left[\sum_{j=1}^2 X_j^e \psi_j^e(x) \right] c_e e^{-\tilde{\lambda}_e k t}, \end{aligned} \quad (2.16)$$

where $c_e = c_n$ are the same coefficients of orthogonality as (2.14) from the Fourier series approximation at element $e = n$, $n = 1, 2, 3$, and X^e is the Eigenvector associated with Eigenvalue $\tilde{\lambda}_e$.

Results derived using initial condition $f(x) = 1$ in (2.9) for an eight-term Fourier series approximation (2.13, 2.14) and an 8-element finite element approximation analogous to (2.16 – 2.14) are generated with Code Listing 2.6. The resulting Eigenfunctions are plotted in Figure 2.6 and resulting approximations in Figure 2.7.

Code Listing 2.6: FEniCS code for solution of the Eigenvalue problem.

```
from pylab import *
from fenics import *
import sympy as sp

n = 8 # number of vertices in mesh
l = 1 # length of the domain
mesh = IntervalMesh(n, 0, l) # 1D mesh
Q = FunctionSpace(mesh, 'CG', 1) # linear-Lagrange F.S.
phi = TestFunction(Q)
u = TrialFunction(Q)
tol = 1e-14

def left(x, on_boundary): return on_boundary and abs(x[0]) < tol
def right(x, on_boundary): return on_boundary and abs(x[0] - l) < tol

gamma_l = DirichletBC(Q, 0.0, left) # left boundary condition
gamma_r = DirichletBC(Q, 0.0, right) # right boundary condition

k = inner(grad(phi), grad(u)) * dx # weak 2nd derivative matrix
m = phi * u * dx # mass matrix

K = PETScMatrix() # container for stiffness matrix
M = PETScMatrix() # container for mass matrix

K = assemble(k, tensor=K) # global assembly of stiffness mat.
M = assemble(m, tensor=M) # global assembly of mass mat.

gamma_r.apply(K) # apply right b.c. to K
gamma_l.apply(K) # apply left b.c. to K
gamma_r.apply(M) # apply right b.c. to M
gamma_l.apply(M) # apply left b.c. to M

eigensolver = SLEPcEigenSolver(K, M) # create solver for Kx = lambda Mx
eigensolver.parameters['solver'] = 'lapack'
eigensolver.solve() # solve with LAPACK

# generate the results and calculate the Fourier coef's with SymPy
x = sp.symbols('x')
xm = mesh.coordinates()[0, :].tolist()
xf = linspace(0, l, 1000)
col = ['k', 'r', '#80cbc4']
lam_m = []
lam_f = []
Am = []
Af = []
Av = []
cn = []

for i in range(n-1):
    ep = eigensolver.get_eigenpair(n-2-i)
    Afi = zeros(n)
    lam_fi = ((i+1) * pi)**2
    Afi = np.sin(np.sqrt(lam_fi) * xf)
    cni = 2/l * sp.integrate(sp.sin((i+1)*pi*x/l), (x, 0, l))
    lam_m.append(ep[0])
    lam_f.append(lam_fi)
    e_v = ep[2].array()
    if i == 1:
        e_v = -e_v
    Am.append(e_v)
    Af.append(Afi)
    cn.append(cni)

# plotting :
=====
mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\\usepackage{fouriernc}']

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

for i, (am, af, lf, lm, c) in enumerate(zip(Am, Af, lam_f, lam_m, col)):
    lbl = r'$n=%i, \lambda=%.2f$' % (i+1, lf)
    ax.plot(xm, am, c, ls='-', lw=2.0, label=lbl)
    lbl = r'$n=%i, \widetilde{\lambda}=%.2f$' % (i+1, lm)
    ax.plot(xf, af, c, ls='--', lw=2.0, label=lbl)

leg = ax.legend(loc='lower left', handlelength=3, fontsize='x-small')
leg.get_frame().set_alpha(0.0)
ax.set_ylim([-1, 1, 1, 1])
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$u$')
ax.grid()
tight_layout()
savefig('../images/fenics_intro/eigenvectors.pdf')

cn = array(cn, dtype='d')
af = zeros(1000)
am = zeros(n+1)
for i in range(n-1):
    af += cn[i]*Af[i]
    am += cn[i]*Am[i]

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

ax.plot(xf, af, 'k--', lw=2.0, label=r'$u(x)$ - Fourier')
ax.plot(xm, am, 'k-', lw=2.0, label=r'$\widetilde{u}(x)$ - FEM')

leg = ax.legend(loc='center')
leg.get_frame().set_alpha(0.0)
```

```
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$u$')
ax.grid()
tight_layout()
savefig('../images/fenics_intro/eigen_solution.pdf')
```

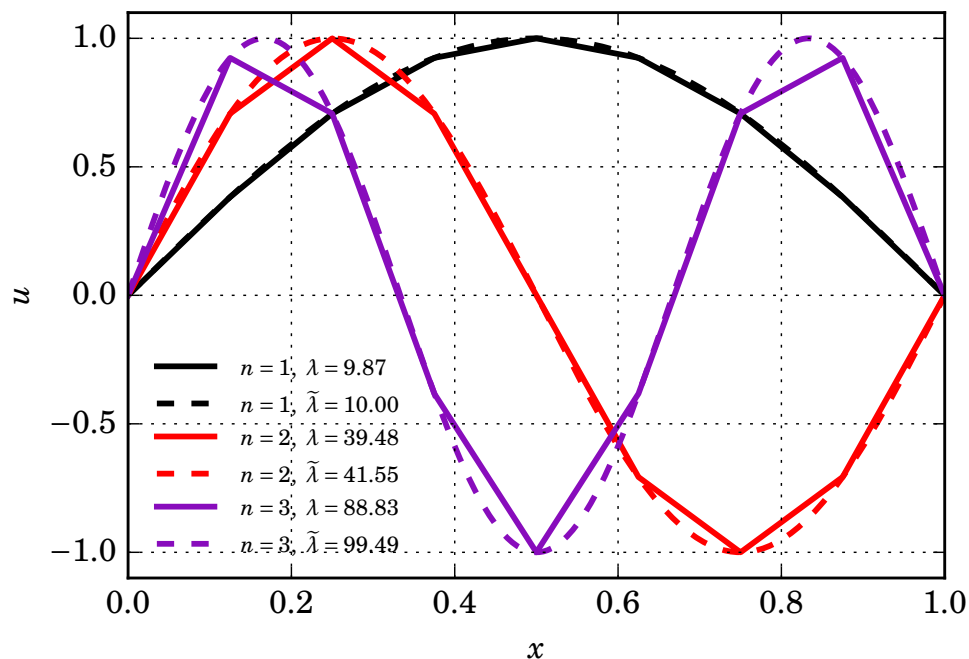


Figure 2.6: Fourier series Eigenfunctions (dashed) and 8 linear element finite element Eigenvectors (solid) for $n = 1, 2, 3$. The associated exact Eigenvalue λ and FEM-approximated Eigenvalues $\tilde{\lambda}$ are listed in the legend.

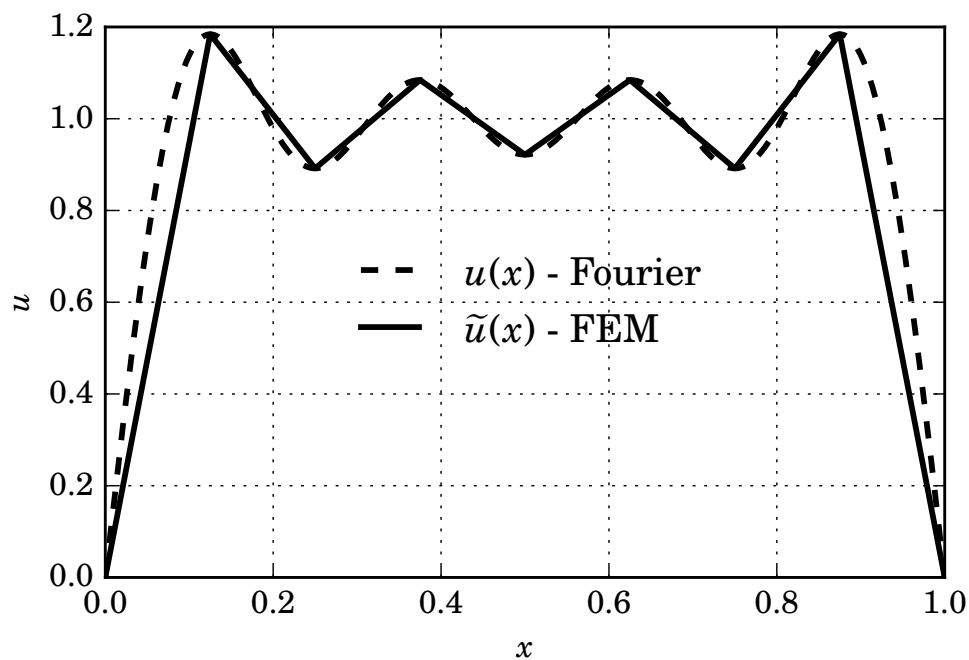


Figure 2.7: Eight-term Fourier series approximation (dashed) and eight-element finite element approximation (solid) to transient problem (2.7 – 2.9) with $\ell = 1$ with $f(x) = 1$ at $t = 0$.

Chapter 3

Problems in two dimensions

For two-dimensional equations, the domain is $\Omega \in \mathbb{R}^2$ and Sobolev space (1.12) becomes

$$\mathcal{H}^1(\Omega) = \left\{ u : \Omega \rightarrow \mathbb{R} \mid u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y} \in L^2(\Omega) \right\}. \quad (3.1)$$

The two-dimensional domain is discretized into either triangular or quadrilateral elements, with new interpolation functions that satisfy a two-dimensional analog of interpolation properties (1.19). For an excellent explanation of the resulting 2D Galerkin system analogous to (1.24), see Elman, Silvester, and Wathen (2005).

In this chapter, we solve the two-dimensional variant of heat equation (1.7) and the Stokes equations.

3.1 Poisson equation

The Poisson equation to be solved over the domain $\Omega = [0, 1] \times [0, 1]$ is

$$-\nabla^2 u = f \quad \text{in } \Omega \quad (3.2)$$

$$f = 10 \sin\left(\frac{2\pi x}{L}\right) \sin\left(\frac{2\pi y}{L}\right) \quad \text{in } \Omega \quad (3.3)$$

$$\nabla u \cdot \mathbf{n} = g_N = \sin(x) \quad \text{on } \Gamma_N, \Gamma_S \quad (3.4)$$

$$u = g_D = 1 \quad \text{on } \Gamma_E, \Gamma_W, \quad (3.5)$$

where Γ_N , Γ_S , Γ_E , and Γ_W are the North, South, East and West boundaries, $L = 1$ is the length of the square side, and \mathbf{n} is the outward normal to the boundary Γ .

The associated Galerkin weak form with test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ defined by (1.11) is

$$\begin{aligned} - \int_{\Omega} \nabla^2 u \phi d\Omega &= \int_{\Omega} f \phi d\Omega \\ \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega - \int_{\Gamma} \phi \nabla u \cdot \mathbf{n} d\Gamma &= \int_{\Omega} f \phi d\Omega \\ \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega - \int_{\Gamma_N} \phi g_N d\Gamma_N - \int_{\Gamma_S} \phi g_N d\Gamma_S &= \int_{\Omega} f \phi d\Omega, \end{aligned}$$

and so the variational problem consists of finding $u \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space 1.10) such that

$$a(u, \phi) = L(\phi) \quad \forall \phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega),$$

subject to Dirichlet condition (3.5), where

$$\begin{aligned} L(\phi) &= \int_{\Gamma_N} \phi g_N d\Gamma_N + \int_{\Gamma_S} \phi g_N d\Gamma_S + \int_{\Omega} f \phi d\Omega, \\ a(u, \phi) &= \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega. \end{aligned}$$

This form is all that is required to derive an approximate solution with FEniCS, as demonstrated by Code Listing 3.1 and Figure 3.1.

Code Listing 3.1: FEniCS solution to two-dimensional Poisson problem (3.2 – 3.5).

```
from fenics import *

# Create mesh and define function space
mesh = UnitSquareMesh(32, 32)
V = FunctionSpace(mesh, "Lagrange", 1)
ff = FacetFunction("size_t", mesh, 0)

# Iterate through the facets and mark each if on a boundary :
#
# 1 - West
# 2 - East
# 3 - North
# 4 - South
for f in facets(mesh):
    n = f.normal() # unit normal vector to facet f
    tol = DOLFIN_EPS
    if n.x() <= -tol and n.y() < tol and f.exterior():
        ff[f] = 1
    elif n.x() >= tol and n.y() < tol and f.exterior():
        ff[f] = 2
    elif n.x() < tol and n.y() >= tol and f.exterior():
        ff[f] = 3
    elif n.x() < tol and n.y() <= -tol and f.exterior():
        ff[f] = 4

ds = Measure('ds')[ff]
dN = ds(3)
dS = ds(4)
dE = ds(2)
dW = ds(1)

# Define boundary condition
u0 = Constant(1.0)
bcE = DirichletBC(V, u0, ff, 2)
bcW = DirichletBC(V, u0, ff, 1)
bc = [bcE, bcW]

# Define variational problem
u = TrialFunction(V)
v = TestFunction(V)
f = Expression("10 * sin(2*pi*x[0]) * sin(2*pi*x[1])")
g = Expression("sin(x[0])")
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*(dN + dS)

# Compute solution
u = Function(V)
solve(a == L, u, bc)

# get array components and triangulation :
v = u.compute_vertex_values(mesh)
x = mesh.coordinates()[0, :]
y = mesh.coordinates()[1, :]
t = mesh.cells()

from pylab import *
from mpl_toolkits.axes_grid1 import make_axes_locatable

fig = figure(figsize=(8,7))
ax = fig.add_subplot(111)

cm = get_cmap('viridis')
c = ax.tricontourf(x, y, t, v, 10, cmap=cm)
p = ax.triplot(x, y, t, '-.', color='k', lw=0.2, alpha=0.4)
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = colorbar(c, cax=cax)
```

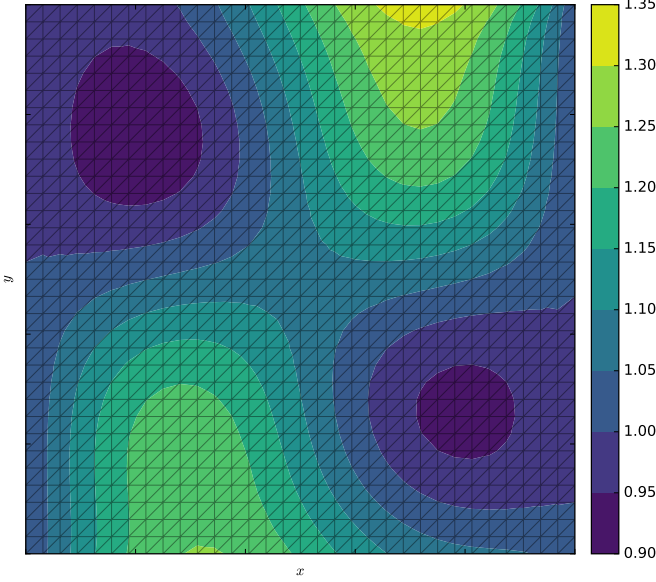


Figure 3.1: Poisson solution for a uniform 32×32 node mesh.

```
tight_layout()
savefig('../images/feics_intro/2Dpoisson.pdf')
```

3.2 Stokes equations with no-slip boundary conditions

The Stokes equations for incompressible fluid over the domain $\Omega = [0, 1] \times [0, 1]$ are

$$-\nabla \cdot \sigma = \mathbf{f} \quad \text{in } \Omega \quad \leftarrow \text{conservation of momentum} \quad (3.6)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega \quad \leftarrow \text{conservation of mass}, \quad (3.7)$$

where σ is the Cauchy-stress tensor defined as $\sigma = 2\eta\dot{\epsilon} - pI$; viscosity η , strain-rate tensor

$$\begin{aligned} \dot{\epsilon} &= \frac{1}{2} [\nabla \mathbf{u} + (\nabla \mathbf{u})^T] \\ &= \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} \end{bmatrix}; \end{aligned} \quad (3.8)$$

velocity \mathbf{u} with components u, v in the x and y directions, and pressure p ; and vector of internal forces \mathbf{f} . For our example, we take $\mathbf{f} = \mathbf{0}$ and boundary conditions

$$\mathbf{u} = \mathbf{g}_D = \mathbf{0} \quad \text{on } \Gamma_N, \Gamma_S, \Gamma_D \quad (3.9)$$

$$\mathbf{u} = \mathbf{g}_E = [-\sin(\pi y) \ 0]^T \quad \text{on } \Gamma_E \quad (3.10)$$

$$\sigma \cdot \mathbf{n} = \mathbf{g}_N = [g_{N_x} \ g_{N_y}]^T = \mathbf{0} \quad \text{on } \Gamma_W, \quad (3.11)$$

where $\Gamma_E, \Gamma_W, \Gamma_N$, and Γ_S are the East, West, North, and South boundaries, Γ_D is the dolphin boundary (Figure 3.2) and \mathbf{n} is the outward-pointing normal vector to these faces. For obvious reasons, velocity boundary condition (3.9) is referred to as a *no-slip* boundary.

It may be of interest to see how the conservation of momentum equations look in their expanded form,

$$\begin{aligned} -\nabla \cdot \sigma &= \mathbf{f} \\ \begin{bmatrix} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} \\ \frac{\partial \sigma_{yx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} \end{bmatrix} &= \begin{bmatrix} 0 \\ 0 \end{bmatrix}, \end{aligned}$$

providing two equations,

$$\begin{aligned} \frac{\partial}{\partial x} \left[2\eta \frac{\partial u}{\partial x} \right] - \frac{\partial p}{\partial x} + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right] &= 0 \\ \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \frac{\partial v}{\partial y} \right] - \frac{\partial p}{\partial y} &= 0, \end{aligned}$$

which along with conservation of mass equation (3.7), also referred to as the *incompressibility constraint*

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0,$$

makes three equations with three unknowns u, v , and p .

Neumann condition (3.11) may be expanded into

$$2\eta \dot{\epsilon}_{xx} n_x - p n_x + 2\eta \dot{\epsilon}_{xy} n_y = g_{N_x} \quad (3.12)$$

$$2\eta \dot{\epsilon}_{yx} n_x + 2\eta \dot{\epsilon}_{yy} n_y - p n_y = g_{N_y}. \quad (3.13)$$

The pressure boundary condition on the outflow boundary Γ_W may be discovered by integrating boundary condition (3.12) along Γ_W with $\mathbf{n} = [-1 \ 0]^T$, assuming a constant viscosity η , and strain-rate tensor definition (3.8),

$$-2\eta \int_{\Gamma_W} \frac{\partial u}{\partial x} d\Gamma_W + \int_{\Gamma_W} p d\Gamma_W = \int_{\Gamma_W} g_{N_x} d\Gamma_W.$$

Next, constraint (3.7) implies that $\partial_x u = -\partial_y v$ and thus

$$\begin{aligned} 2\eta \int_0^1 \frac{\partial v}{\partial y} dy + \int_0^1 p dy &= \int_{\Gamma_W} g_{N_x} d\Gamma_W \\ 2\eta v(0, 1) - 2\eta v(0, 0) + \int_0^1 p dy &= \int_{\Gamma_W} g_{N_x} d\Gamma_W \\ \int_0^1 p dy &= \int_{\Gamma_W} 0 d\Gamma_W, \end{aligned}$$

which implies that $p = 0$ over the entire outflow boundary Γ_W (for further illustration, see Elman, Silvester, and Wathen (2005)).

The weak form for problem (3.6, 3.7, 3.9 – 3.11) is formed by taking the inner product of both sides of the conservation of momentum equation with the vector test function $\Phi = [\phi \ \psi]^T \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^2$ (see test space (1.11)) integrating over the domain Ω ,

$$-\int_{\Omega} \nabla \cdot \sigma \cdot \Phi d\Omega = \int_{\Omega} \mathbf{f} \cdot \Phi d\Omega,$$

then integrate by parts to get

$$\begin{aligned} \int_{\Omega} \sigma : \nabla \Phi d\Omega - \int_{\Gamma} \sigma \cdot \mathbf{n} \cdot \Phi d\Gamma &= \int_{\Omega} \mathbf{f} \cdot \Phi d\Omega \\ \int_{\Omega} \sigma : \nabla \Phi d\Omega &= \int_{\Omega} \mathbf{f} \cdot \Phi d\Omega, \end{aligned}$$

where the facts that $\sigma \cdot \mathbf{n} = \mathbf{0}$ on the West boundary and Dirichlet conditions exist on the North, South, East, and dolphin boundaries has been used. Next, taking the inner product of incompressibility (conservation of mass) equation (3.7) with the test function $\xi \in M^h \subset L^2(\Omega)$ (see L^2 space (1.13)) integrating over Ω ,

$$\int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega = 0.$$

Finally, using the fact that the right-hand side of incompressibility equation (3.7) is zero, the *mixed formulation* (see for example Johnson (2009)) consists of finding *mixed approximation* $\mathbf{u} \in \mathbf{S}_E^h \subset (\mathcal{H}_E^1(\Omega))^2$ and $p \in M^h \subset L^2(\Omega)$ such that

$$a(\mathbf{u}, p, \Phi, \xi) = L(\Phi) \quad \forall \Phi \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^2, \quad \xi \in M^h \subset L^2(\Omega), \quad (3.14)$$

subject to Dirichlet conditions (3.9 – 3.11) and

$$\begin{aligned} a(\mathbf{u}, p, \Phi, \xi) &= \int_{\Omega} \sigma : \nabla \Phi \, d\Omega + \int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega, \\ L(\Phi) &= \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega. \end{aligned}$$

3.2.1 Stability

In order to derive a unique solution for pressure p , the trial and test spaces must be chosen in such a way that the *inf-sup condition*

$$\min_{\xi \neq \text{constant}} \left\{ \max_{\Phi \neq \mathbf{0}} \left\{ \frac{|(\xi, \nabla \cdot \Phi)|}{\|\Phi\|_{1,\Omega} \|\xi\|_{0,\Omega}} \right\} \right\} \geq \gamma \quad (3.15)$$

is satisfied for any conceivable grid and some constant γ (Elman, Silvester, and Wathen, 2005). The notation $(f, g) = \int_{\Omega} f g \, d\Omega$ is the inner product, $\|\mathbf{f}\|_{1,\Omega} = \left(\int_{\Omega} [\mathbf{f} \cdot \mathbf{f} + \nabla \mathbf{f} : \nabla \mathbf{f}] \, d\Omega \right)^{1/2}$ is the \mathbf{S}_E^h -norm, and $\|f\|_{0,\Omega} = \left(\int_{\Omega} f^2 \, d\Omega \right)^{1/2}$ is a so-called *quotient space norm* (Elman, Silvester, and Wathen, 2005).

One way of satisfying inf-sup condition (3.15) is through the use of Taylor-Hood finite element space (Taylor and Hood, 1973), which utilize a quadratic function space for the velocity vector components and the linear Lagrange function space for the pressure.

The velocity and pressure solutions to (3.14) using Taylor-Hood elements are depicted in Figure 3.2 and generated by Code Listing 3.2.

3.3 Stokes equations with slip-friction boundary conditions

A *slip-friction* boundary condition for Stokes equations (3.6, 3.7) using an identical domain Ω as in §3.2 may be generated by replacing no-slip boundary condition (3.9) with the pair of boundary conditions

$$\mathbf{u} \cdot \mathbf{n} = g_D \quad \text{on } \Gamma_N, \Gamma_S, \Gamma_D \quad (3.16)$$

$$(\sigma \cdot \mathbf{n})_{\parallel} = \mathbf{g}_N = -\beta \mathbf{u} \quad \text{on } \Gamma_N, \Gamma_S, \Gamma_D, \quad (3.17)$$

where $(\mathbf{v})_{\parallel} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$ denotes the tangential component of a vector \mathbf{v} and $\beta \geq 0$ is a friction coefficient. Boundary conditions (3.16, 3.17) are equivalent to no-slip boundary condition (3.9) as β approaches infinity. Note also that impenetrability condition (3.16) specifies one component of velocity and is an essential boundary condition, while friction condition (3.17) specifies the other component (in three dimension it would specify the other two components) and is a natural boundary condition. For comparison purposes, we use the same inflow boundary condition (3.10) and outflow boundary condition (3.11).

The weak form for problem (3.6, 3.7, 3.16, 3.17, 3.10 3.11) is formed by taking the inner product of both sides of the conservation of momentum equation with the vector test function $\Phi = [\phi \, \psi]^T \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^2$ (see test space (1.11)) integrating over the domain Ω ,

$$-\int_{\Omega} \nabla \cdot \sigma \cdot \Phi \, d\Omega = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega,$$

then integrate by parts to get and add the incompressibility constraint as performed in §3.2,

$$\int_{\Omega} \sigma : \nabla \Phi \, d\Omega - \int_{\Gamma} \sigma \cdot \mathbf{n} \cdot \Phi \, d\Gamma + \int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega. \quad (3.18)$$

Expanding tangential stress condition (3.17), we have

$$\sigma \cdot \mathbf{n} = (\mathbf{n} \cdot \sigma \cdot \mathbf{n})\mathbf{n} - \beta \mathbf{u},$$

producing

$$\mathcal{B}_{\Omega} + \mathcal{B}_{\Gamma_G} + \mathcal{B}_{\Gamma_E} = \mathcal{F} \quad (3.19)$$

with individual terms

$$\begin{aligned} \mathcal{B}_{\Omega} &= + \int_{\Omega} \sigma(\mathbf{u}, p) : \nabla \Phi \, d\Omega + \int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega \\ \mathcal{B}_{\Gamma_G} &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\mathbf{u}, p) \cdot \mathbf{n}) \mathbf{n} \cdot \Phi \, d\Gamma_G + \int_{\Gamma_G} \beta \mathbf{u} \cdot \Phi \, d\Gamma_G \\ \mathcal{B}_{\Gamma_E} &= - \int_{\Gamma_E} \sigma(\mathbf{u}, p) \cdot \mathbf{n} \cdot \Phi \, d\Gamma_E \\ \mathcal{F} &= + \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega, \end{aligned}$$

where $\Gamma_G = \Gamma_N \cup \Gamma_S \cup \Gamma_D$ is the entire slip-friction boundary, and the fact that $\sigma \cdot \mathbf{n} = \mathbf{0}$ on the West boundary Γ_W has been used.

A method devised by Nitsche, 1970/71 and further explained by Freund and Stenberg, 1995 imposes Dirichlet conditions (3.10, 3.16) in a weak form by adjoining symmetric terms to (3.19),

$$\mathcal{B}_{\Omega} + \mathcal{B}_{\Gamma_G} + \mathcal{B}_{\Gamma_G}^W + \mathcal{B}_{\Gamma_E} + \mathcal{B}_{\Gamma_E}^W = \mathcal{F} + \mathcal{F}^W, \quad (3.20)$$

where

$$\begin{aligned}\mathcal{B}_{\Gamma_G}^W &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) \mathbf{n} \cdot \mathbf{u} \, d\Gamma_G + \gamma \int_{\Gamma_G} \frac{1}{h} (\mathbf{u} \cdot \mathbf{n}) (\Phi \cdot \mathbf{n}) \, d\Gamma_G \\ \mathcal{B}_{\Gamma_E}^W &= - \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{u} \, d\Gamma_E + \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{u}) \, d\Gamma_E \\ \mathcal{F}^W &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) g_D \, d\Gamma_G + \gamma \int_{\Gamma_G} \frac{1}{h} g_D \Phi \cdot \mathbf{n} \, d\Gamma_G \\ &\quad - \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{g}_D \, d\Gamma_E + \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{g}_D) \, d\Gamma_E.\end{aligned}$$

with element diameter h and application-specific parameter $\gamma > 0$ normally derived by experimentation. Variational form (3.20) is justified using the properties of the self-adjoint linear differential operator σ :

$$\begin{aligned}\int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\mathbf{u}, p) \cdot \mathbf{n}) \mathbf{n} \cdot \Phi \, d\Gamma_G &= \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) \mathbf{n} \cdot \mathbf{u} \, d\Gamma_G \\ \int_{\Gamma_E} \sigma(\mathbf{u}, p) \cdot \mathbf{n} \cdot \Phi \, d\Gamma_E &= \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{u} \, d\Gamma_E,\end{aligned}$$

and using boundary conditions (3.16, 3.17),

$$\begin{aligned}\int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) \mathbf{n} \cdot \mathbf{u} \, d\Gamma_G &= \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) g_D \, d\Gamma_G \\ \int_{\Gamma_E} \sigma(\mathbf{u}, p) \cdot \mathbf{n} \cdot \Phi \, d\Gamma_E &= \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{g}_D \, d\Gamma_E.\end{aligned}$$

The extra terms

$$\begin{aligned}\gamma \int_{\Gamma_G} \frac{1}{h} (\mathbf{u} \cdot \mathbf{n}) (\Phi \cdot \mathbf{n}) \, d\Gamma_G &= \gamma \int_{\Gamma_G} \frac{1}{h} g_D \Phi \cdot \mathbf{n} \, d\Gamma_G \\ \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{u}) \, d\Gamma_E &= \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{g}_D) \, d\Gamma_E\end{aligned}$$

have been added to enable the simulator to enforce boundary conditions (3.16, 3.17) to the desired level of accuracy.

Finally, the mixed formulation consistent with problem (3.6, 3.7, 3.16, 3.17, 3.10, 3.11) reads: find mixed approximation $\mathbf{u} \in \mathbf{S}_E^h \subset (\mathcal{H}_E^1(\Omega))^2$ and $p \in M^h \subset L^2(\Omega)$ subject to (3.20) for all $\Phi \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^2$ and $\xi \in M^h \subset L^2(\Omega)$.

Identically to §3.3, we use the Taylor-Hood element to satisfy inf-sup condition (3.15). The friction along the dolphin, North, and South boundaries was taken to be $\beta = 10$, and the Nitsche parameter $\gamma = 100$ was derived by experimentation. The velocity \mathbf{u} and pressure p solutions to this problem are depicted in Figure 3.3 and were generated from Code Listing 3.3.

Code Listing 3.2: FEniCS code used to solve 2D-Stokes-no-slip problem (3.14).

```
from fenics import *

# load mesh and subdomains :
mesh = Mesh("meshes/dolphin_fine.xml.gz")
sub_domains = MeshFunction("size_t", mesh,
                           "meshes/dolphin_fine_subdomains.xml.gz")

# define Taylor-Hood function space "W" :
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V * Q

# no-slip boundary condition for velocity
# y = 0, y = 1 and around the dolphin :
noslip = Constant((0, 0))
bc0 = DirichletBC(W.sub(0), noslip, sub_domains, 0)
```

```
# inflow boundary condition for velocity at x = 1 :
inflow = Expression(("sin(x[1]*pi)", "0.0"))
bc1 = DirichletBC(W.sub(0), inflow, sub_domains, 1)

# collect boundary conditions :
bcs = [bc0, bc1]

# viscosity :
eta = 1.0

# identity tensor :
I = Identity(2)

# strain-rate tensor :
def epsilon(u): return 0.5*(grad(u) + grad(u).T)

# Cauchy stress tensor :
def sigma(u, p): return 2*eta*epsilon(u) - p*I

# define variational problem :
u, p = TrialFunctions(W)
v, q = TestFunctions(W)

f = Constant((0, 0))
a = inner(sigma(u, p), grad(v)) * dx + q * div(u) * dx
L = inner(f, v) * dx

# compute solution :
w = Function(W)
solve(a == L, w, bcs)

# split the mixed solution using deepcopy
# (needed for further computation on coefficient vector)
u, p = w.split(True)

print "Norm of velocity coefficient vector: %.15g" % u.vector().norm("l2")
print "Norm of pressure coefficient vector: %.15g" % p.vector().norm("l2")

# get individual components with deep copy :
u0, u1 = u.split(True)

from pylab import *
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import colors, ticker

# calculate array componets :
v0 = u0.compute_vertex_values(mesh)
v1 = u1.compute_vertex_values(mesh)
v = sqrt(v0**2 + v1**2 + 1e-16)
v0 = v0 / v
v1 = v1 / v
x = mesh.coordinates()[0, :]
y = mesh.coordinates()[1, :]
t = mesh.cells()

# generate velocity figure :
fig = figure(figsize=(8, 7))
ax = fig.add_subplot(111)

v[v > 2.0] = 2.0
cm = get_cmap('viridis')
ls = array([0.0, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.00001])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-.', color='k', lw=0.2, alpha=0.3)
q = ax.quiver(x, y, v0, v1, pivot='middle',
              color='k',
              scale=60,
              width=0.0015,
              headwidth=4.0,
              headlength=4.0,
              headaxislength=4.0)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = fig.colorbar(c, cax=cax, ticks=ls, format='%.1f')
tight_layout()
savefig('.../images/fenics_intro/2DStokes_u.pdf')

# generate pressure figure :
v = p.compute_vertex_values(mesh)

fig = figure(figsize=(8, 7))
ax = fig.add_subplot(111)

v[v > 120] = 120
v[v < -20] = -20

ls = array([v.min(), -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, 10, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-.', color='k', lw=0.2, alpha=0.5)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = colorbar(c, cax=cax)
tight_layout()
savefig('.../images/fenics_intro/2DStokes_p.pdf')
```

Code Listing 3.3: FEniCS code used to approximate the solution to the 2D-Stokes-slip-friction problem of §3.3.

```
from fenics import *

mesh = Mesh("meshes/dolphin_fine.xml.gz")
sub_domains = MeshFunction("size_t", mesh,
                           "meshes/dolphin_fine_subdomains.xml.gz")

# Taylor-Hood element
V = VectorFunctionSpace(mesh, 'CG', 2)
Q = FunctionSpace(mesh, 'CG', 1)
W = V * Q
```



```

# variational problem :
u, p = TrialFunctions(W)
v, q = TestFunctions(W)

# no penetration boundary condition for velocity
# y = 0, y = 1 and around the dolphin :
u_n = Constant(0.0)

# inflow boundary condition for velocity at x = 1 :
u_0 = Expression((" -sin(x[1]*pi)", "0.0"))

# relevant measures :
ds = Measure("ds")[sub_domains]
dG_0 = ds(0)
dG_r = ds(1)

# constants :
gamma = Constant(1e2)
h = CellSize(mesh)
n = FacetNormal(mesh)
I = Identity(2)
eta = Constant(1.0)
f = Constant((0.0,0.0))
beta = Constant(10.0)

def epsilon(u): return 0.5*(grad(u) + grad(u).T)
def sigma(u,p): return 2*eta*epsilon(u) - p*I

t = dot(sigma(u,p), n)
s = dot(sigma(v,q), n)

B_o = + inner(sigma(u,p), grad(v))*dx - div(u)*q*dx

B_g = - dot(n,t) * dot(v,n) * dG_0 \
      - dot(u,n) * dot(s,n) * dG_0 \
      + gamma/h * dot(u,n) * dot(v,n) * dG_0 \
      + beta * dot(u, v) * dG_0 \
      - inner(dot(sigma(u,p), n), v) * dG_r \
      - inner(dot(sigma(v,q), n), u) * dG_r \
      + gamma/h * inner(v,u) * dG_r

F = + dot(f,v) * dx \
    + gamma/h * u_n * dot(v,n) * dG_0 \
    - inner(dot(sigma(v,q), n), u_0) * dG_r \
    + gamma/h * inner(v,u_0) * dG_r

# solve variational problem
wh = Function(W)

solve(B_o + B_g == F, wh)
uh, ph = wh.split(True)

print "Norm of velocity coefficient vector: %.15g" % uh.vector().norm("l2")
print "Norm of pressure coefficient vector: %.15g" % ph.vector().norm("l2")

# get individual components with deep copy :
u0,u1 = uh.split(True)

from pylab import *
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import colors, ticker

# calculate array componets :
v0 = u0.compute_vertex_values(mesh)
v1 = u1.compute_vertex_values(mesh)
v = sqrt(v0**2 + v1**2 + 1e-16)
v0 = v0 / v
v1 = v1 / v
x = mesh.coordinates()[0,:]
y = mesh.coordinates()[1,:]
t = mesh.cells()

# generate velocity figure :
fig = figure(figsize=(8,7))
ax = fig.add_subplot(111)

v[v > 2.0] = 2.0
cm = get_cmap('viridis')
ls = array([0.0,0.1,0.2,0.4,0.6,0.8,1.0,1.2,1.4,1.6,1.8,2.00001])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-', color='k', lw=0.2, alpha=0.3)
q = ax.quiver(x, y, v0, v1, pivot='middle',
             color='k',
             scale=60,
             width=0.0015,
             headwidth=4.0,
             headlength=4.0,
             headaxislength=4.0)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = fig.colorbar(c, cax=cax, ticks=ls, format='%.1f')
tight_layout()
savefig('.../images/fenics_intro/2Dstokes_nitsche_u.pdf')

# generate pressure figure :
v = ph.compute_vertex_values(mesh)

fig = figure(figsize=(8,7))
ax = fig.add_subplot(111)

v[v > 120] = 120
v[v < -20] = -20

ls = array([v.min(), -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, 10, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-', color='k', lw=0.2, alpha=0.5)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.set_xticklabels([])
ax.set_yticklabels([])
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = colorbar(c, cax=cax)
tight_layout()
savefig('.../images/fenics_intro/2Dstokes_nitsche_p.pdf')

```

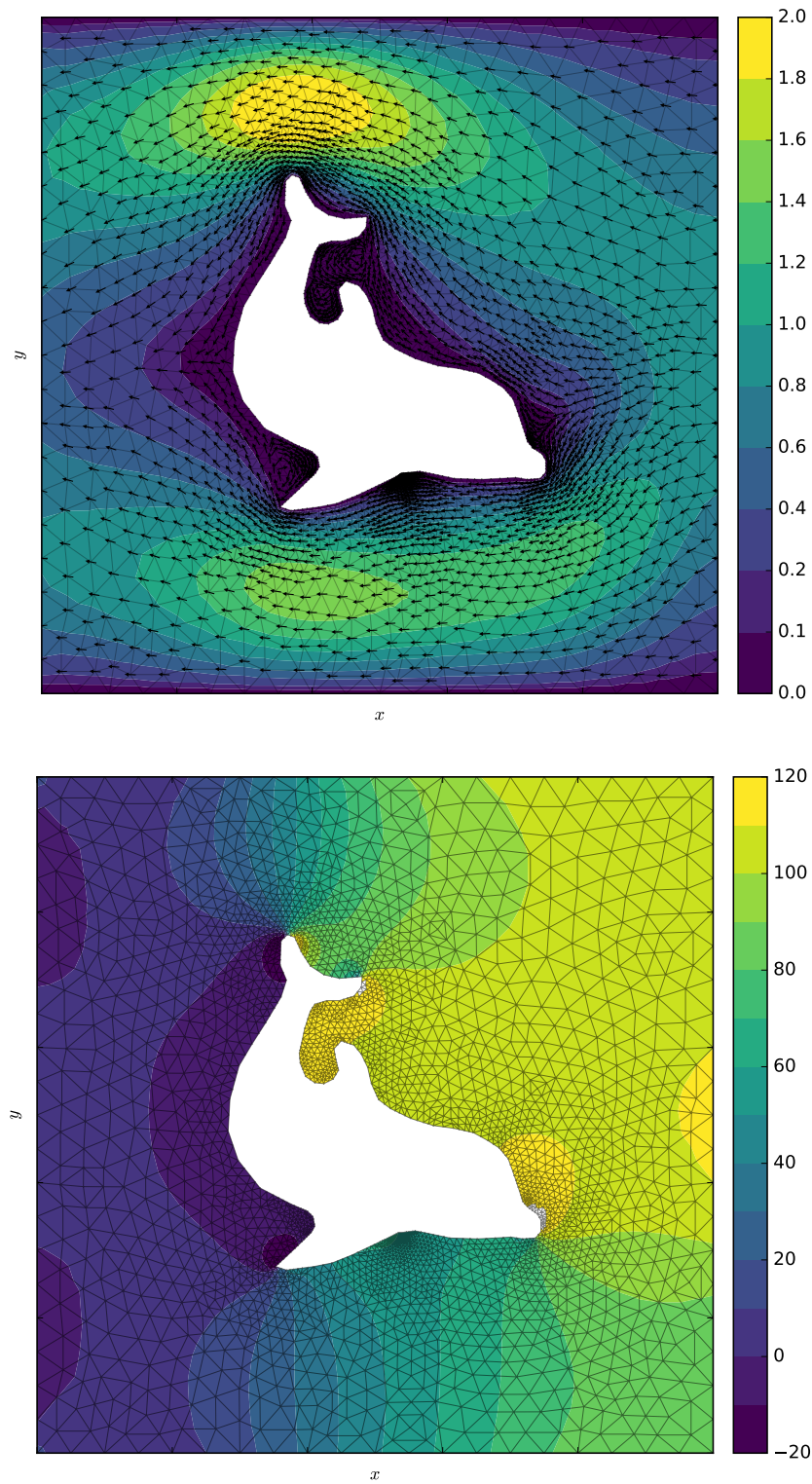


Figure 3.2: Velocity field \mathbf{u} (top) and pressure p (bottom) for the no-slip formulation given in §3.2 using the Taylor-hood element (referred to as the P2 – P1 approximation).

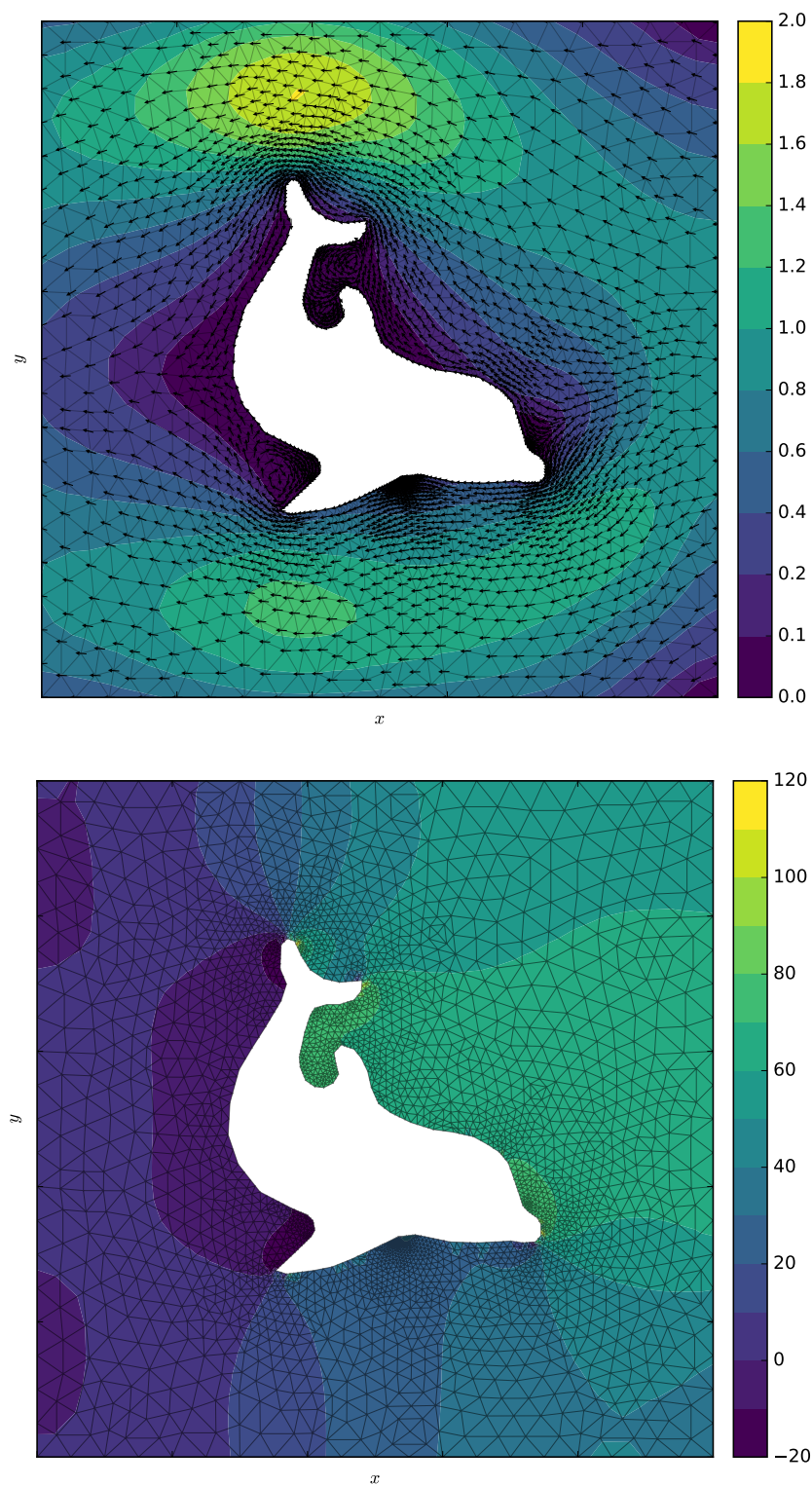


Figure 3.3: Velocity field \mathbf{u} (top) and pressure p (bottom) for the slip-friction formulation given in §3.3 using the same Taylor-Hood element as model §3.2, with results depicted in Figure 3.2.

Chapter 4

Problems in three dimensions

For three dimensional equations, the domain of the system $\Omega \in \mathbb{R}^3$ and Sobolev space (1.12) becomes

$$\mathcal{H}^1(\Omega) = \left\{ u : \Omega \rightarrow \mathbb{R} \mid u, \frac{\partial u}{\partial x}, \frac{\partial u}{\partial y}, \frac{\partial u}{\partial z} \in L^2(\Omega) \right\}, \quad (4.1)$$

and the domain is discretized into either tetrahedral or brick elements, with new interpolation functions that satisfy 3D interpolation properties analogous to (1.19). For an excellent explanation of the resulting 3D Galerkin system analogous to (1.24), see Elman, Silvester, and Wathen (2005).

In this chapter, we solve the three-dimensional variant of heat equation (1.7) and the Stokes equations.

4.1 Poisson equation

The Poisson equation to be solved over the domain $\Omega = [-\pi, \pi] \times [-\pi, \pi] \times [-\pi, \pi]$ is

$$-\nabla^2 u = f \quad \text{in } \Omega \quad (4.2)$$

$$f = 10 \exp\left(-\frac{x^2}{2} - \frac{y^2}{2} - \frac{z^2}{2}\right) \quad \text{in } \Omega \quad (4.3)$$

$$\nabla u \cdot \mathbf{n} = g_{N_y} = \sin(x) \quad \text{on } \Gamma_N, \Gamma_S \quad (4.4)$$

$$\nabla u \cdot \mathbf{n} = g_{N_x} = 0 \quad \text{on } \Gamma_E, \Gamma_W \quad (4.5)$$

$$u = g_D = 0 \quad \text{on } \Gamma_T, \Gamma_B, \quad (4.6)$$

where Γ_N , Γ_S , Γ_E , and Γ_W are the North, South, East and West boundaries, Γ_T and Γ_B are the top and bottom boundaries, and \mathbf{n} is the outward unit normal to the boundary Γ .

The associated Galerkin weak form with test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ defined by (1.11) is

$$\begin{aligned} - \int_{\Omega} \nabla^2 u \phi d\Omega &= \int_{\Omega} f \phi d\Omega \\ \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega - \int_{\Gamma} \phi \nabla u \cdot \mathbf{n} d\Gamma &= \int_{\Omega} f \phi d\Omega \\ \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega - \int_{\Gamma_N} \phi g_{N_y} d\Gamma_N - \int_{\Gamma_S} \phi g_{N_y} d\Gamma_S &= \int_{\Omega} f \phi d\Omega, \end{aligned}$$

and so the discrete variational problem consists of finding $u \in S_E^h \subset \mathcal{H}_{E_0}^1(\Omega)$ given by (1.10) such that

$$a(u, \phi) = l(\phi) \quad \forall \phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega),$$

subject to Dirichlet condition (4.6), where

$$\begin{aligned} l(\phi) &= \int_{\Gamma_N} \phi g_{N_y} d\Gamma_N + \int_{\Gamma_S} \phi g_{N_y} d\Gamma_S + \int_{\Omega} f \phi d\Omega, \\ a(u, \phi) &= \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega. \end{aligned}$$

This is as far as is needed to proceed in order to solve this simple problem with FEniCS, as demonstrated by Code Listing 4.1 and Figures 4.1 and 4.2.

Code Listing 4.1: FEniCS code used to solve 3D-Poisson problem (4.2 – 4.6).

```
from fenics import *

# create mesh :
p1 = Point(-pi, -pi, -pi)
p2 = Point(pi, pi, pi)
mesh = BoxMesh(p1, p2, 30, 30, 30)

# define function space :
V = FunctionSpace(mesh, "Lagrange", 1)

# create a MeshFunction for marking boundaries :
ff = FacetFunction("size_t", mesh, 0)

# iterate through the facets and mark each if on a boundary :
# 1 = ..... top | 4 = ..... West side
# 2 = ..... bottom | 5 = ..... North side
# 3 = ..... East side | 6 = ..... South side
for f in facets(mesh):
    n = f.normal() # unit normal vector to facet f
    tol = 1e-10
    if n.z() >= tol and f.exterior():
        ff[f] = 1
    elif n.z() <= -tol and f.exterior():
        ff[f] = 2
    elif n.x() > tol and n.y() < tol and f.exterior():
        ff[f] = 3
    elif n.x() < -tol and n.y() < tol and f.exterior():
        ff[f] = 4
    elif n.y() > tol and n.x() < tol and f.exterior():
        ff[f] = 5
    elif n.y() < -tol and n.x() < tol and f.exterior():
        ff[f] = 6

# need the N and S boundary for natural conditions :
ds = Measure("ds")[ff]
dN = ds(5) + ds(6)

# define essential boundary conditions :
zero = Constant(0.0)
bcN = DirichletBC(V, zero, ff, 2)
bcS = DirichletBC(V, zero, ff, 1)

# define variational functions :
u = TrialFunction(V)
v = TestFunction(V)

# expressions for known data :
f = Expression("10 * exp(-(pow(x[0],2)/2 + pow(x[1],2)/2 + pow(x[2],2)/2))")
g = Expression("sin(x[0])")

# variational problem :
a = inner(grad(u), grad(v))*dx
L = f*v*dx + g*v*dN

# compute solution :
u = Function(V)
solve(a == L, u, [bcN, bcS])

File("output/u.pvd") << u
```


4.2 Stokes equations

Recall from §3.2 that the Stokes equations for an incompressible fluid are

$$-\nabla \cdot \sigma = \mathbf{f} \quad \text{in } \Omega \quad \leftarrow \text{conservation of momentum} \quad (4.7)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega \quad \leftarrow \text{conservation of mass}, \quad (4.8)$$

where σ is the Cauchy-stress tensor defined as $\sigma = 2\eta\dot{\epsilon} - pI$; viscosity η , strain-rate tensor

$$\begin{aligned} \dot{\epsilon} &= \frac{1}{2} [\nabla \mathbf{u} + (\nabla \mathbf{u})^T] \\ &= \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} & \frac{1}{2} \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \\ \frac{1}{2} \left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \right) & \frac{1}{2} \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) & \frac{\partial w}{\partial z} \end{bmatrix}, \end{aligned} \quad (4.9)$$

velocity $\mathbf{u} = [u \ v \ w]^T$ with components u, v, w in the x, y , and z directions, and pressure p ; and vector of internal forces $\mathbf{f} = \rho \mathbf{g}$ composed of material density ρ and gravitational acceleration vector $\mathbf{g} = [0 \ 0 \ -g]^T$. For our example we use boundary conditions

$$\sigma \cdot \mathbf{n} = \mathbf{g}_N = \mathbf{0} \quad \text{on } \Gamma_T, \Gamma_B \quad (4.10)$$

$$\mathbf{u} = \mathbf{g}_D = \mathbf{0} \quad \text{on } \Gamma_L, \quad (4.11)$$

where Γ_T, Γ_B , and Γ_L are the top, bottom, and lateral boundaries and \mathbf{n} is the outward-pointing normal vector.

It may be of interest to see how the conservation of momentum equations look in their expanded form,

$$-\nabla \cdot \sigma = \mathbf{f}$$

$$\begin{bmatrix} \frac{\partial \sigma_{xx}}{\partial x} + \frac{\partial \sigma_{xy}}{\partial y} + \frac{\partial \sigma_{xz}}{\partial z} \\ \frac{\partial \sigma_{yx}}{\partial x} + \frac{\partial \sigma_{yy}}{\partial y} + \frac{\partial \sigma_{yz}}{\partial z} \\ \frac{\partial \sigma_{zx}}{\partial x} + \frac{\partial \sigma_{zy}}{\partial y} + \frac{\partial \sigma_{zz}}{\partial z} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \\ \rho g \end{bmatrix}$$

so that we have three equations,

$$\frac{\partial}{\partial x} \left[2\eta \frac{\partial u}{\partial x} \right] - \frac{\partial p}{\partial x} + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial x} + \frac{\partial w}{\partial z} \right) \right] = 0 \quad (4.12a)$$

$$\frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \frac{\partial v}{\partial y} \right] - \frac{\partial p}{\partial y} + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} \right) \right] = 0 \quad (4.12b)$$

$$\frac{\partial}{\partial x} \left[\eta \left(\frac{\partial w}{\partial z} + \frac{\partial u}{\partial x} \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial w}{\partial z} + \frac{\partial v}{\partial y} \right) \right] + \frac{\partial}{\partial z} \left[2\eta \frac{\partial w}{\partial z} \right] - \frac{\partial p}{\partial z} = \rho g, \quad (4.12c)$$

which when combined with the conservation of mass equation,

$$\nabla \cdot \mathbf{u} = \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0,$$

gives four equations and four unknowns u, v, w , and p .

The weak form for this problem is formed by taking the inner product of both sides of conservation of momentum equation (4.7) with the vector test function $\Phi = [\phi \ \psi \ \chi]^T \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^3$ (see test space (1.11)) integrating over the domain Ω ,

$$-\int_{\Omega} \nabla \cdot \sigma \cdot \Phi \, d\Omega = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega,$$

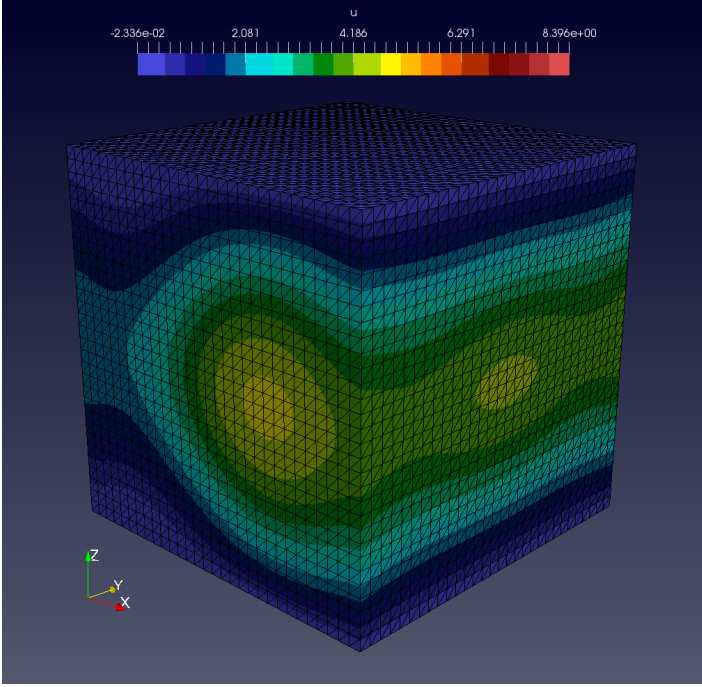


Figure 4.1: Poisson equation solution to (4.2 – 4.6) defined over a $30 \times 30 \times 30$ element mesh.

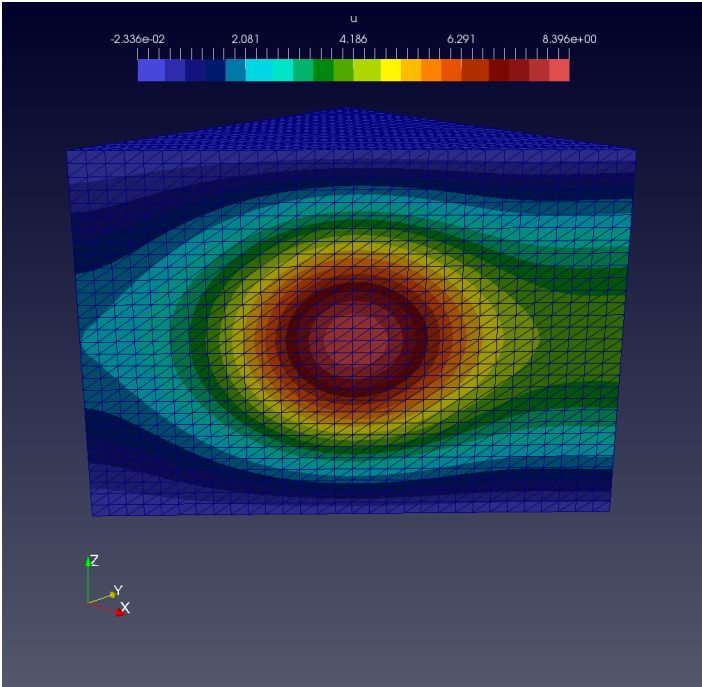


Figure 4.2: Inside the Poisson solution represented by Figure 4.1.

then integrate by parts to get

$$\int_{\Omega} \sigma : \nabla \Phi \, d\Omega - \int_{\Gamma} \sigma \cdot \mathbf{n} \cdot \Phi \, d\Gamma = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega$$

$$\int_{\Omega} \sigma : \nabla \Phi \, d\Omega = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega,$$

where the fact that all boundaries are either homogeneous Neumann or Dirichlet has been used to eliminate boundary integrals. Next, multiplying incompressibility (conservation of mass) equation (4.8) by the test function $\xi \in M^h \subset L^2(\Omega)$ (see L^2 space (1.13)) also integrating over Ω ,

$$\int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega = 0.$$

Finally, using the fact that the right-hand side of incompressibility equation (4.8) is zero, the mixed formulation consists of finding mixed approximation $\mathbf{u} \in \mathbf{S}_E^h \subset (\mathcal{H}_E^1(\Omega))^3$ (see trial space (1.10)) and $p \in M^h \subset L^2(\Omega)$ such that

$$a(\mathbf{u}, p, \Phi, \xi) = L(\Phi) \quad \forall \mathbf{u} \in \mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^3, \quad \xi \in M^h \subset L^2(\Omega), \quad (4.13)$$

subject to Dirichlet condition (4.11) and

$$a(\mathbf{u}, p, \Phi, \xi) = \int_{\Omega} \sigma : \nabla \Phi \, d\Omega + \int_{\Omega} (\nabla \cdot \mathbf{u}) \xi \, d\Omega,$$

$$L(\Phi) = \int_{\Omega} \mathbf{f} \cdot \Phi \, d\Omega.$$

For our solution satisfying inf-sup condition (3.15), we enrich the finite element space with bubble functions (see Chapter 5), thus creating MINI elements (Arnold, Brezzi, and Fortin, 1984). The solution is generated with Code Listing 4.2 and depicted in Figure 4.3.

Code Listing 4.2: FEniCS solution to 3D-Stokes-no-slip problem (4.13).

```
from fenics import *

mesh = Mesh('meshes/unit_cyl_mesh.xml')

# Define function spaces
#B = FunctionSpace(mesh, "B", 4)
#Q = FunctionSpace(mesh, "CG", 1)
#M = Q + B
#V = MixedFunctionSpace([M, M, M])
#W = MixedFunctionSpace([V, Q])
V = VectorFunctionSpace(mesh, "CG", 2)
Q = FunctionSpace(mesh, "CG", 1)
W = V + Q
ff = FacetFunction('size_t', mesh, 0)

# iterate through the facets and mark each if on a boundary :
#
# 1 = high slope, upward facing ..... top
# 2 = high slope, downward facing ..... bottom
# 3 = low slope, upward or downward facing ..... side
for f in facets(mesh):
    n = f.normal() # unit normal vector to facet f
    tol = 1.0
    if n.z() >= tol and f.exterior():
        ff[f] = 1
    elif n.z() <= -tol and f.exterior():
        ff[f] = 2
    elif abs(n.z()) < tol and f.exterior():
        ff[f] = 3

L = 1.0/2000.0
xmin = -L
xmax = L
ymin = -L
ymax = L

# Deform the mesh to the defined geometry :
for x in mesh.coordinates():
    # transform x :
    x[0] = x[0] * (xmax - xmin)
    # transform y :
    x[1] = x[1] * (ymax - ymin)
    # transform z :
    # thickness = surface - base, z = thickness + base
    x[2] = x[2] * 5.0/1000.0

# constants :
```

```
rho = Constant(1420.0)
eta = Constant(8.0)
g = Constant(9.8)
x = SpatialCoordinate(mesh)
n = FacetNormal(mesh)
I = Identity(3)

# =====
# define variational problem :
U = TrialFunction(W)
tst = TestFunction(W)

u, p = split(U)
v, q = split(tst)

# no-slip boundary condition for velocity :
bc = DirichletBC(W.sub(0), Constant((0, 0, 0)), ff, 3)

# stress and strain tensors :
def sigma(u, p): return 2*eta*epsilon(u) - p*I
def epsilon(u): return 0.5*(grad(u) + grad(u).T)

# internal force vector :
f = rho * as_vector([0, 0, -g])

# conservation of momentum :
a = + inner(sigma(u, p), grad(v)) * dx \
    + div(u) * q * dx
L = + dot(f, v) * dx

# solve the linear system :
U = Function(W)
solve(a == L, U, bc, solver_parameters = {"linear_solver" : "minres",
                                           "preconditioner" : "hybre_amg"})

u, p = U.split(True)

File("output/u.pvd") << u
File("output/p.pvd") << p
```

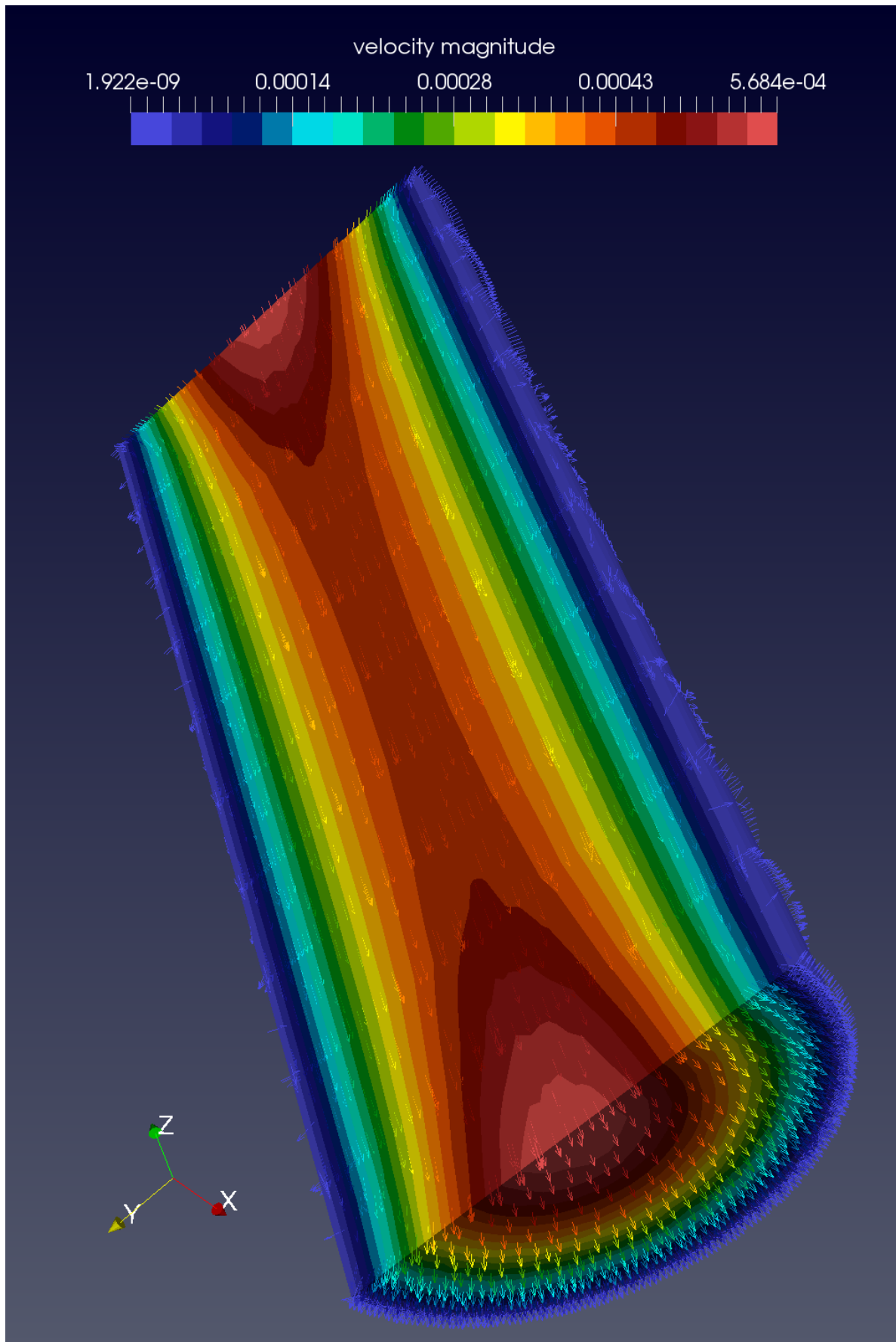


Figure 4.3: Inside Stokes velocity solution \mathbf{u} in m s^{-1} within a tube of diameter 1 mm and height 5 mm filled with honey at density $\rho = 1420 \text{ kg m}^{-3}$ and viscosity $\eta = 8 \text{ Pa s}$. Gravity forces the honey in the $-z$ direction, as indicated by the overlain velocity vectors.

Chapter 5

Subgrid scale effects

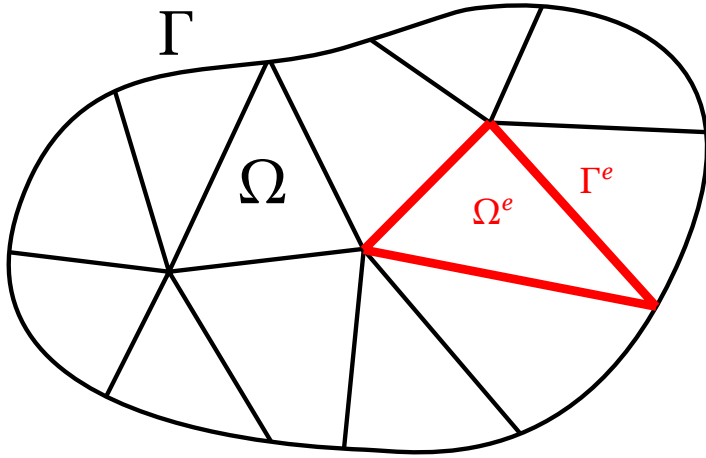


Figure 5.1: The domain of the partitioned problem.

The discretization of a domain for which a differential equation is to be solved often results in numerically unstable solutions. These instabilities result from *subgrid*-scale effects that cannot be accounted for at the resolution of the reduced domain. Here we review several techniques for stabilizing the solution to such problems through the development of a distributional formulation including an approximation of these subgrid-scale effects.

5.1 Subgrid scale models

As presented by Hughes (1995), consider the bounded domain Ω with boundary Γ discretized into N element subdomains Ω^e with boundaries Γ^e (Figure 5.1). Let

$$\begin{aligned}\Omega' &= \bigcup_{e=1}^N \Omega^e && \leftarrow \text{element interiors} \\ \Gamma' &= \bigcup_{e=1}^N \Gamma^e && \leftarrow \text{element boundaries} \\ \Omega &= \bar{\Omega} = \text{closure}(\Omega').\end{aligned}$$

The abstract problem that we wish to solve consists of finding a function $u \in L^2(\Omega)$, such that for given functions

$$f \in L^2(\Omega), q \in L^2(\Gamma),$$

$$\mathcal{L}u = f \quad \text{in } \Omega \quad (5.1)$$

$$u = q \quad \text{on } \Gamma, \quad (5.2)$$

where \mathcal{L} is a possibly non-symmetric differential operator and the unknown u is composed of overlapping resolvable scales \bar{u} and unresolvable, or subgrid scales u' , i.e. $u = \bar{u} + u'$.

The variational form of this system may be stated using the definition of the adjoint \mathcal{L}^* of the operator \mathcal{L}

$$a(w, u) = (w, \mathcal{L}u) = (\mathcal{L}^*w, u) \quad (5.3)$$

for all sufficiently smooth u, w such that

$$u = q, \quad w = 0 \quad \text{on } \Gamma,$$

where

$$u = \bar{u} + u' \quad \text{and} \quad w = \bar{w} + w'.$$

Making the assumption that the unresolvable scales vanish on element boundaries,

$$u' = w' = 0 \quad \text{on } \Gamma',$$

variational Equation (5.3) is transformed into

$$\begin{aligned}a(w, u) &= (w, f) \\ a(\bar{w} + w', \bar{u} + u') &= (\bar{w} + w', f) \\ a(\bar{w}, \bar{u}) + a(\bar{w}, u') + a(w', \bar{u}) + a(w', u') &= (\bar{w}, f) + (w', f).\end{aligned}$$

providing two equations, which we collect in terms of \bar{w} and w' ,

$$\begin{cases} a(\bar{w}, \bar{u}) + a(\bar{w}, u') = (\bar{w}, f) \\ (\bar{w}, \mathcal{L}\bar{u}) + (\bar{w}, \mathcal{L}u') = (\bar{w}, f) \\ (\bar{w}, \mathcal{L}\bar{u}) + (\mathcal{L}^*\bar{w}, u') = (\bar{w}, f) \end{cases} \quad (5.4)$$

and

$$\begin{cases} a(w', \bar{u}) + a(w', u') = (w', f) \\ (w', \mathcal{L}\bar{u}) + (w', \mathcal{L}u') = (w', f). \end{cases} \quad (5.5)$$

The Euler-Lagrange equations corresponding to second sub-problem (5.5) are

$$\begin{aligned} \mathcal{L}\bar{u} + \mathcal{L}u' &= f \\ \implies \mathcal{L}u' &= -(\mathcal{L}\bar{u} - f) && \text{in } \Omega^e \\ u' &= 0 && \text{on } \Gamma^e. \end{aligned} \quad (5.6)$$

Thus, the differential operator \mathcal{L} applied to the unresolvable scales u' is equal to the residual of the resolved scales, $f - \mathcal{L}\bar{u}$ when we assume that the unresolvable scales vanish on element boundaries.

5.2 Green's function for \mathcal{L}

The Green's function problem for a linear operator \mathcal{L} in problem (5.1) seeks to find $g(x, y)$ such that

$$u(y) = (\mathcal{L}^{-1}f)(y) = \int_{\Omega'} g(x, y) f(x) d\Omega_x,$$

with Green's functions satisfying

$$\begin{aligned} \mathcal{L}g &= \delta && \text{in } \Omega^e \\ g &= 0 && \text{on } \Gamma^e, \end{aligned}$$

where δ is the Dirac delta distribution. An expression for the unresolvable scales may be formed in terms of the resolvable scales from (5.6):

$$u'(y) = - \int_{\Omega'} g(x, y) (\mathcal{L}\bar{u} - f)(x) d\Omega'_x. \quad (5.7)$$

Substituting this expression into (5.4) results in

$$(\bar{w}, \mathcal{L}\bar{u}) + (\mathcal{L}^* \bar{w}, M(\mathcal{L}\bar{u} - f)) = (\bar{w}, f), \quad (5.8)$$

where

$$Mv(x) = - \int_{\Omega'} g(x, y) v(y) d\Omega'_y \quad (5.9)$$

and

$$(\mathcal{L}^* \bar{w}, M(\mathcal{L}\bar{u} - f)) = - \int_{\Omega'} \int_{\Omega'} (\mathcal{L}^* \bar{w})(y) g(x, y) (\mathcal{L}\bar{u} - f)(x) d\Omega_x d\Omega_y.$$

Expression (5.8) may be stated in the bilinear form

$$B(\bar{w}, \bar{u}; g) = L(\bar{w}; g),$$

where

$$\begin{aligned} B(\bar{w}, \bar{u}; g) &= (\bar{w}, \mathcal{L}\bar{u}) + (\mathcal{L}^* \bar{w}, M\mathcal{L}\bar{u}) \\ L(\bar{w}; g) &= (\mathcal{L}^* \bar{w}, Mf) + (\bar{w}, f). \end{aligned}$$

Thus all the effects of the unresolvable scales have been accounted for up to the assumption that u' vanish on element boundaries. Next is derived an approximation of Green's function g and a development of a finite-dimensional analog of (5.8).

5.3 Bubbles

The space of bubble functions consists of the set of functions that vanish on element boundaries and whose maximum values is one, the space

$$\mathcal{B}_0^k(\Omega) = \left\{ u \in \mathcal{H}^k(\Omega) \mid u = 0 \text{ on } \Gamma^e, \|u\|_\infty = 1 \right\}. \quad (5.10)$$

For a concrete example, the lowest order – corresponding to $k = 2$ in (5.10) – one-dimensional reference bubble function is defined as

$$\phi_e'(x) = 4\psi_1^e(x)\psi_2^e(x), \quad (5.11)$$

with basis given by the one-dimensional linear Lagrange interpolation functions described previously in §1.2.3,

$$\psi_1^e(x) = 1 - \frac{x}{h_e} \quad \psi_2^e(x) = \frac{x}{h_e},$$

where h_e is the width of element e . This basis satisfies the required interpolation properties

$$\psi_i^e(x_j) = \delta_{ij} \quad \sum_{j=1}^n \psi_j^e(x) = 1,$$

where n is the number of element equations. Note that ϕ' has the properties that $\|\phi'\|_\infty = 1$ and is zero on the element boundaries (Code Listing 5.1 and Figure 5.2).

The lowest order – corresponding to $k = 2$ in (5.10) – two-dimensional triangular reference element bubble function is defined as

$$\phi_{2e}'(x, y) = 27\psi_1^{2e}(x, y)\psi_2^{2e}(x)\psi_3^{2e}(y), \quad (5.12)$$

with basis given by the quadratic Lagrange interpolation functions

$$\psi_1^{2e}(x, y) = 1 - \frac{x}{h_x} - \frac{y}{h_y}, \quad \psi_2^{2e}(x) = \frac{x}{h_e}, \quad \psi_3^{2e}(y) = \frac{y}{h_e},$$

where h_x is the max height in the x direction and h_y is the max height of in the y direction of the reference element e . Again, $\phi_{2e}'(x, y)$ has the properties that $\|\phi_{2e}'\|_\infty = 1$ and is zero on the element boundaries (Code Listing 5.2 and Figure 5.3).

Code Listing 5.1: Python code used to generate Figure 5.2.

```
from pylab import *
x = linspace(0,1,1000)
phi_1 = 1 - x
phi_2 = x
mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]
fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)
ax.set_ylim(0.0, 1.10)
ax.plot(x, phi_1, 'k-', lw=2.0, label=r'$\psi_1$')
ax.plot(x, phi_2, 'k--', lw=2.0, label=r'$\psi_2$')
ax.plot(x, 4*phi_1*phi_2, 'r', lw=2.0, label=r'$\phi$')
leg = ax.legend(loc='lower center')
leg.get_frame().set_alpha(0.0)
ax.set_xlabel(r'$x$')
ax.grid()
tight_layout()
savefig("../images/bubbles/bubble_new.pdf")
```

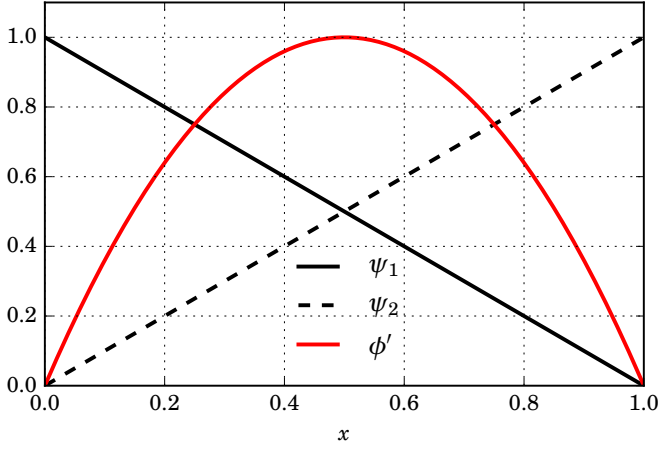


Figure 5.2: The lowest order one-dimensional triangular element reference bubble function given by (5.11) with $h_e = 1$.

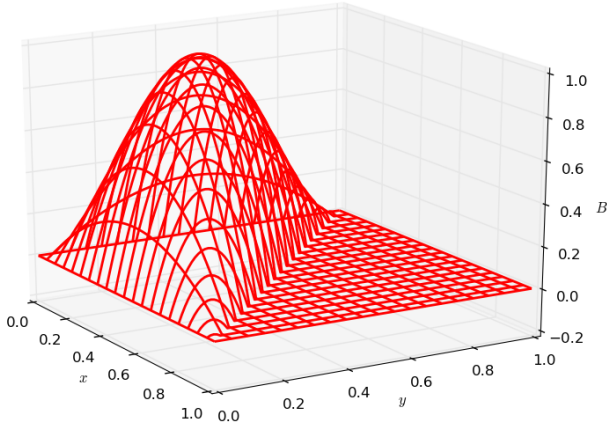


Figure 5.3: The lowest order two-dimensional reference bubble function given by (5.12) with $h_x = h_y = 1$.

Code Listing 5.2: Python code used to generate Figure 5.3.

```
from pylab import *
from mpl_toolkits.mplot3d import Axes3D

def B(x,y):
    return 27 * (1-x-y) * x * y

x = linspace(0,1,100)
y = linspace(0,1,100)
X,Y = meshgrid(x,y)

# plot the results :
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

Z = B(X,Y)

Z[X + Y > 1] = 0 # zero the area outside the triangle

ax.plot_wireframe(X, Y, Z, color='r', lw=2.0, rstride=5, cstride=5)

ax.set_ylabel(r'$y$')
ax.set_xlabel(r'$x$')
ax.set_zlabel(r'$B$')
tight_layout()
show()
```

5.4 Approximation of Green's function for \mathcal{L} with bubbles

Following the work of Hughes (1995), if $\phi'_j(x)$, $j \in 1, 2, \dots, N_b$ is a set of N_b linearly independent bubble functions, a single element's unresolvable scales can be approximated – in a process referred to as *static condensation* – by linearly expanding u' into N_b nodes and \bar{u} into N_n nodes,

$$u'(x) \approx u'_h(x) = \sum_{j=1}^{N_b} \phi'_j(x) u'_j, \quad \bar{u}(x) \approx \bar{u}_h(x) = \sum_{j=1}^{N_n} \psi_j(x) \bar{u}_j, \quad (5.13)$$

where u'_j is the coefficient associated with bubble function j and \bar{u}_j is the coefficient associated with a finite-element shape function j .

Fixing $w'_k = \phi'_k$ and inserting $u'_h(x)$ into (5.5),

$$\begin{aligned} a(w'_k, \bar{u}_h) + a(w'_k, u'_h) &= (w'_k, f) \\ a(\phi'_k, \bar{u}_h) + a\left(\phi'_k, \sum_{j=1}^{N_b} \phi'_j u'_j\right) &= (\phi'_k, f) \\ \left(\phi'_k, \mathcal{L}\left(\sum_{j=1}^{N_b} \phi'_j u'_j\right)\right) &= (\phi'_k, f) - (\phi'_k, \mathcal{L}\bar{u}_h) \\ \left(\phi'_k, \sum_{j=1}^{N_b} \mathcal{L}\phi'_j u'_j\right) &= -(\phi'_k, \mathcal{L}\bar{u}_h - f) \\ \sum_{j=1}^{N_b} \left(\phi'_k, \mathcal{L}\phi'_j\right) u'_j &= -(\phi'_k, \mathcal{L}\bar{u}_h - f) \\ \sum_{j=1}^{N_b} a(\phi'_k, \phi'_j) u'_j &= -(\phi'_k, \mathcal{L}\bar{u}_h - f), \end{aligned}$$

for $k = 1, 2, \dots, N_b$. This implies that the j nodal values of $u'_h(x)$ in (5.13) are given by the linear system of equations

$$u'_j = - \sum_{k=1}^{N_b} a(\phi'_j, \phi'_k)^{-1} (\phi'_k(x), \mathcal{L}\bar{u}_h(x) - f(x)),$$

where $a(\phi'_j, \phi'_k)^{-1}$ is the jk component of the inverse of matrix $a(\phi'_j, \phi'_k)$. Inserting this into $u'_h(y)$ given by (5.13),

$$\begin{aligned} u'_h(y) &= \sum_{j=1}^{N_b} \phi'_j(y) u'_j \\ &= \sum_{j=1}^{N_b} \phi'_j(y) \left[- \sum_{k=1}^{N_b} a(\phi'_j, \phi'_k)^{-1} (\phi'_k(x), \mathcal{L}\bar{u}_h(x) - f(x)) \right] \\ &= - \sum_{j,k=1}^{N_b} \phi'_j(y) \left[a(\phi'_j, \phi'_k)^{-1} (\phi'_k(x), \mathcal{L}\bar{u}_h(x) - f(x)) \right] \\ &= - \sum_{j,k=1}^{N_b} \int_{\Omega'} \phi'_j(y) \left[a(\phi'_j, \phi'_k)^{-1} \right] \phi'_k(x) (\mathcal{L}\bar{u}_h - f)(x) d\Omega'_x \\ &= - \int_{\Omega'} \left(\sum_{j,k=1}^{N_b} \phi'_j(y) \left[a(\phi'_j, \phi'_k)^{-1} \right] \phi'_k(x) \right) (\mathcal{L}\bar{u}_h - f)(x) d\Omega'_x \\ &= - \int_{\Omega'} \tilde{g}(x, y) (\mathcal{L}\bar{u}_h - f)(x) d\Omega'_x, \end{aligned}$$

thus providing the Green's function approximation in (5.7)

$$g(x, y) \approx \tilde{g}(x, y) = \sum_{j,k=1}^{N_b} \phi'_j(y) \left[a \left(\phi'_j, \phi'_k \right)^{-1} \right] \phi'_k(x). \quad (5.14)$$

Finally, inserting resolvable scale approximation \bar{u}_h defined by (5.13) and Green's function approximation (5.14) into (5.8), we have

$$(\bar{w}_h, \mathcal{L}\bar{u}_h) + (\mathcal{L}^* \bar{w}_h, \tilde{M}(\mathcal{L}\bar{u}_h - f)) = (\bar{w}_h, f), \quad (5.15)$$

where

$$Mv(x) \approx \tilde{M}v(x) = - \int_{\Omega'} \tilde{g}(x, y) v(x) d\Omega'_x,$$

and the associated approximate bilinear form

$$B(\bar{w}_h, \bar{u}_h; \tilde{g}) = L(\bar{w}_h; \tilde{g}),$$

where

$$\begin{aligned} B(\bar{w}_h, \bar{u}_h; \tilde{g}) &= (\bar{w}_h, \mathcal{L}\bar{u}_h) + (\mathcal{L}^* \bar{w}_h, \tilde{M}\mathcal{L}\bar{u}_h) \\ L(\bar{w}_h; \tilde{g}) &= (\mathcal{L}^* \bar{w}_h, \tilde{M}f) + (\bar{w}_h, f). \end{aligned}$$

5.5 Stabilized methods

As described by Hughes (1995) and Codina (1998), stabilized methods are *generalized Galerkin methods* of the form

$$(\bar{w}_h, \mathcal{L}\bar{u}_h) + (\mathbb{L}\bar{w}_h, \tau(\mathcal{L}\bar{u}_h - f)) = (\bar{w}_h, f), \quad (5.16)$$

where operator \mathbb{L} is a differential operator typically chosen from

$$\mathbb{L} = +\mathcal{L} \quad \text{Galerkin/least-squares (GLS)} \quad (5.17)$$

$$\mathbb{L} = +\mathcal{L}_{\text{adv}} \quad \text{SUPG} \quad (5.18)$$

$$\mathbb{L} = -\mathcal{L}^* \quad \text{subgrid-scale model (SSM)} \quad (5.19)$$

where \mathcal{L}_{adv} is the advective part of the operator \mathcal{L} .

Note that when using differential operator (5.19), stabilized form (5.16) implies that $\tau = -\tilde{M} \approx -M$, and therefore the *intrinsic-time* parameter τ approximates integral operator (5.9). Equivalently,

$$\tau \cdot \delta(y - x) = \tilde{g}(x, y) \approx g(x, y),$$

and we can generate an explicit formula for τ by integrating over a single element Ω^e ,

$$\int_{\Omega^e} \int_{\Omega^e} \tilde{g}(x, y) d\Omega_x^e d\Omega_y^e = \int_{\Omega^e} \int_{\Omega^e} \tau \cdot \delta(y - x) d\Omega_x^e d\Omega_y^e = \tau h,$$

$$\implies \tau = \frac{1}{h} \int_{\Omega^e} \int_{\Omega^e} \tilde{g}(x, y) d\Omega_x^e d\Omega_y^e,$$

where h is the element diameter. Therefore, the parameter τ will depend both on the operator \mathcal{L} and the basis chosen for Green's function approximation \tilde{g} as evident by (5.14).

For example, when \mathcal{L} is the advective-diffusive operator $\mathcal{L}u = -\nabla \cdot \sigma(u) = -\nabla \cdot (k\nabla u - \mathbf{a}u)$ and linear-Lagrange elements are used, the optimal expression for τ is the *stream-line upwind/Petrov-Galerkin* (SUPG) coefficient (Brooks and Hughes, 1982; Hughes, 1995).

$$\tau_{\text{SUPG}} = \frac{h}{2|\mathbf{a}|} \left(\coth(P_\epsilon) - \frac{1}{P_\epsilon} \right), \quad P_\epsilon = \frac{h|\mathbf{a}|}{2\kappa}, \quad (5.20)$$

where P_ϵ is the element Péclet number and \mathbf{a} is the material velocity vector.

On the other hand, if \mathcal{L} is the diffusion-reaction operator $\mathcal{L}u = -\nabla \cdot (k\nabla u) + su$ with absorption coefficient $s \geq 0$, $P_\epsilon = 0$ and τ is given by the coefficient (Hughes, Franca, and Hulbert, 1989)

$$\tau_{\text{DR}} = \alpha \frac{h^2}{\kappa}, \quad (5.21)$$

where α is a mesh-size-independent parameter dependent on the specific model used.

When \mathcal{L} is the advective-diffusion-reaction equation $\mathcal{L}u = -\nabla \cdot (k\nabla u) - \mathbf{a} \cdot \nabla u + su$, Codina (1998) used the coefficient

$$\tau_{\text{ADR}} = \frac{1}{\frac{4\kappa}{h^2} + \frac{2|\mathbf{a}|}{h} + s}, \quad (5.22)$$

to stabilize the formulation over a range of values for s and $|\mathbf{a}|$ using the space of linear Lagrange interpolation functions ψ .

Finally, when \mathcal{L} is the Stokes operator $\mathcal{L}(\mathbf{u}, p) = -\nabla \cdot \sigma(\mathbf{u}, p) = -\nabla \cdot (2\eta \epsilon(\mathbf{u}) - p\mathbf{I})$, τ has been found to be the coefficient (Hughes, Franca, and Balestra, 1986)

$$\tau_{\text{S}} = \alpha \frac{h^2}{2\eta}, \quad (5.23)$$

where the unknowns consist of the material velocity \mathbf{u} and pressure p (see §3.2, §3.3, and §4.2), and $\alpha > 0$ may or may not depend on the basis used for \mathbf{u} .

5.6 Diffusion-reaction problem

For an example, consider the steady-state advection-diffusion equation defined over the domain $\Omega \in [0, 1]$

$$\mathcal{L}u = -\kappa \frac{d^2 u}{dx^2} + su = f, \quad u(0) = 1, \quad u'(1) = 0, \quad (5.24)$$

with diffusion coefficient κ , absorption coefficient $s \geq 0$, and source term f are constant throughout the domain.

5.6.1 Bubble-enriched solution

The bubble-function-enriched distributional form of the equation consists of finding $\hat{u} = \bar{u} + u'$ where $\bar{u} \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space (1.10)) and $u' \in B_0^h \subset \mathcal{B}_0^2(\Omega)$ (see bubble space (5.10))

such that

$$\begin{aligned}
 (\hat{\psi}, \mathcal{L}\hat{u}) &= (\hat{\psi}, f) \\
 -\kappa \int_{\Omega} \frac{d^2 \hat{u}}{dx^2} \hat{\psi} d\Omega + s \int_{\Omega} \hat{u} \hat{\psi} d\Omega &= \int_{\Omega} f \hat{\psi} d\Omega \\
 \kappa \int_{\Omega} \frac{d\hat{u}}{dx} \frac{d\hat{\psi}}{dx} d\Omega - \kappa \int_{\Gamma} \frac{d\hat{u}}{dx} \hat{\psi} d\Gamma + s \int_{\Omega} \hat{u} \hat{\psi} d\Omega &= \int_{\Omega} f \hat{\psi} d\Omega \\
 \kappa \int_{\Omega} \frac{d\hat{u}}{dx} \frac{d\hat{\psi}}{dx} d\Omega + s \int_{\Omega} \hat{u} \hat{\psi} d\Omega &= \int_{\Omega} f \hat{\psi} d\Omega.
 \end{aligned}$$

for all *enriched* or *augmented* test functions $\hat{\psi} = \psi + \phi$, where $\psi \in S_0^h \subset \mathcal{H}_E^1(\Omega)$ (see test space (1.11)) and $\phi \in B_0^h \subset \mathcal{B}_0^2(\Omega)$ (bubble space (5.10)).

5.6.2 SSM-stabilized solution

The subgrid-scale stabilized distributional form of the equation is derived by using subgrid-scale-model operator (5.19) and DR stability parameter (5.21) within the general stabilized form (5.16). Thus, the stabilized problem consists of finding $\tilde{u} \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space (1.10)) such that

$$(\psi, \mathcal{L}\tilde{u}) - (\mathcal{L}^* \psi, \tau_{\text{DR}}(\mathcal{L}\tilde{u} - f)) = (\psi, f),$$

for all test functions $\psi \in S_0^h \subset \mathcal{H}_E^1(\Omega)$ (see test space (1.11)). Using the fact that the diffusion-reaction operator (5.24) is self adjoint, we have the bilinear form

$$B(\tilde{u}, \psi) = L(\psi),$$

where

$$\begin{aligned}
 B(u, \psi) &= \kappa \int_{\Omega} \frac{d\tilde{u}}{dx} \frac{d\psi}{dx} d\Omega + s \int_{\Omega} \tilde{u} \psi d\Omega - \frac{\alpha}{\kappa} \int_{\Omega} h^2 (\mathcal{L}^* \psi) (\mathcal{L}\tilde{u}) d\Omega \\
 L(\psi) &= \int_{\Omega} f \psi d\Omega - \frac{\alpha}{\kappa} \int_{\Omega} h^2 \mathcal{L}^* \psi f d\Omega.
 \end{aligned}$$

Note if linear Lagrange elements are used as a basis for \tilde{u} and ψ , the diffusive terms with coefficient κ in $\mathcal{L}\tilde{u}$ and $\mathcal{L}^* \psi$ will be zero.

5.6.3 Analytic solution

With the constants

$$\kappa = \frac{1}{500}, \quad s = 1, \quad f = 0,$$

the analytic solution to (5.24) is

$$u_a(x) = \frac{\exp(-10\sqrt{5}(x-2)) + \exp(10\sqrt{5}x)}{1 + \exp(20\sqrt{5})}.$$

Note that this is a heavily reaction-dominated problem, resulting in high gradients in the solution, referred to as a *boundary layer*, near $x = 0$. The analytic solution is plotted against solutions determined with the standard Galerkin method with linear Lagrange elements ψ , quadratic-bubble-enriched linear Lagrange elements $\hat{\psi}$, and the SSM-stabilized formulation in Figure 5.4, generated by Code Listing 5.3.

Code Listing 5.3: FEniCS code used solve diffusion-reaction problem (5.24).

```

from fenics import *

mesh = IntervalMesh(10,0,1)
Q = FunctionSpace(mesh, 'CG', 1)
B = FunctionSpace(mesh, 'B', 2)
M = Q + B

def left(x, on_boundary):
    return on_boundary and x[0] == 0

kappa = Constant(1.0/500.0)
s = Constant(1.0)
f = Constant(0.0)

# standard Galerkin solution :
leftBC = DirichletBC(Q, 1.0, left)

u = TrialFunction(Q)
v = TestFunction(Q)
us = Function(Q)
uf1 = Function(Q)

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + s * u * v * dx
L = f * v * dx

solve(a == L, us, leftBC)

uf1.interpolate(us)

# enriched space solution :
uD = project(Constant(1.0), M)
leftBC = DirichletBC(M, uD, left)

u = TrialFunction(M)
v = TestFunction(M)
us = Function(M)
uf2 = Function(Q)

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + s * u * v * dx
L = f * v * dx

solve(a == L, us, leftBC)

uf2.interpolate(us)

# SSM stabilized :
leftBC = DirichletBC(Q, 1.0, left)

u = TrialFunction(Q)
v = TestFunction(Q)
us = Function(Q)
uf3 = Function(Q)
h = CellSize(mesh)
C = Constant(1/15.0)
tau = C * h**2 / kappa

def L(u): return -(kappa * u.dx(0)).dx(0) + s*u

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + s * u * v * dx \
    - inner(L(v), tau*L(u)) * dx
L = f * v * dx \
    - inner(L(v), tau*f) * dx

solve(a == L, us, leftBC)

uf3.interpolate(us)

# plotting :
from pylab import *

purp = '#8B0000'

t = mesh.coordinates()[0,0][:-1]
uf1 = uf1.vector().array()
uf2 = uf2.vector().array()
uf3 = uf3.vector().array()

x = linspace(0, 1, 1000)
ue = (exp(-10*sqrt(5)*(x-2)) + exp(10*sqrt(5)*x))/(1 + exp(20*sqrt(5)))

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}]

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

ax.plot(t, uf1, 'r', ls='-', lw=2.0, label=r"$u$")
ax.plot(t, uf2, 'k', ls='-', lw=2.0, label=r"$\hat{u}$")
ax.plot(t, uf3, purp, ls='--', lw=2.0, label=r"$\tilde{u}$")
ax.plot(x, ue, 'k--', ls='--', lw=2.0, label=r"$u_{\mathrm{a}}$")

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$u$')
ax.grid()
leg = ax.legend(loc='upper right')
leg.get_frame().set_alpha(0.0)
ax.set_xlim([0,0.3])
tight_layout()
savefig('../images/bubbles/DR_analytic_new.pdf')

```

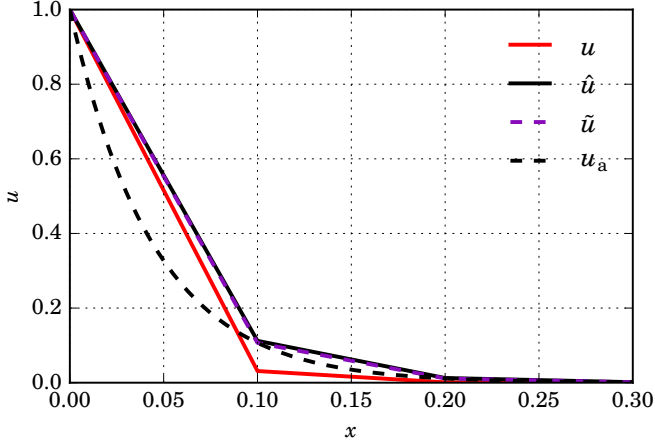


Figure 5.4: The analytic solution u_a (black dashed) plotted against the unstabilized method solution u (red), bubble-enriched solution \hat{u} (dashed purple), and SSM-stabilized solution \tilde{u} (black). The parameter $\alpha = \frac{1}{15}$ was found by experimentation.

5.7 Advection-diffusion-reaction example

Consider the model defined over the domain $\Omega \in [0, 1]$

$$\mathcal{L}u = -\kappa \frac{d^2 u}{dx^2} + d \frac{du}{dx} + su = f, \quad u(0) = 0, \quad u'(1) = 0, \quad (5.25)$$

where κ is the diffusion coefficient, d is the velocity of the material, $s \geq 0$ is an absorption coefficient, and f is a source term.

5.7.1 GLS-stabilized solution

The Galerkin/least-squares stabilized distributional form of the equation is derived by using GLS operator (5.17) and ADR stability parameter (5.22) within the general stabilized form (5.16). Thus, the stabilized problem consists of finding $\tilde{u} \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space (1.10)) such that

$$(\psi, \mathcal{L}\tilde{u}) + (\mathcal{L}\psi, \tau_{\text{ADR}}(\mathcal{L}\tilde{u} - f)) = (\psi, f),$$

for all test functions $\psi \in S_0^h \subset \mathcal{H}_E^1(\Omega)$ (see test space (1.11)). Using ADR stability parameter (5.22), we have the bilinear form

$$B(\tilde{u}, \psi) = L(\psi),$$

where

$$\begin{aligned} B(\tilde{u}, \psi) &= \kappa \int_{\Omega} \frac{d\tilde{u}}{dx} \frac{d\psi}{dx} d\Omega + d \int_{\Omega} \frac{d\tilde{u}}{dx} \psi d\Omega + s \int_{\Omega} \tilde{u} \psi d\Omega \\ &\quad + \int_{\Omega} \tau_{\text{ADR}}(\mathcal{L}\psi)(\mathcal{L}\tilde{u}) d\Omega \\ L(\psi) &= \int_{\Omega} f \hat{\psi} d\Omega + \int_{\Omega} \tau_{\text{ADR}} \mathcal{L}\psi f d\Omega. \end{aligned}$$

Note if linear Lagrange elements are used as a basis for \tilde{u} and ψ , the diffusive terms with coefficient κ in $\mathcal{L}\tilde{u}$ and $\mathcal{L}\psi$ will be zero, and if $s = 0$, the GLS and SUPG operators given by (5.17) and (5.18), respectively, would in this case be identical (Hughes, Franca, and Hulbert, 1989).

For an extreme example, we take

$$\kappa = \frac{1}{100}, \quad d = 10, \quad s = 5,$$

and

$$f = \begin{cases} 1000 & \text{if } x = 0.5 \\ 0 & \text{otherwise} \end{cases},$$

resulting in an equation with low diffusivity that is heavily dominated by gradients of u while advecting u in the $+x$ direction. Solutions determined with the standard Galerkin method with linear Lagrange elements ψ , quadratic-bubble-enriched linear Lagrange elements $\hat{\psi}$, and the GLS-stabilized formulation are depicted in Figure 5.5 and generated by Code Listing 5.4.

Code Listing 5.4: FEniCS code used solve advection-diffusion-reaction problem (5.25).

```
from fenics import *

mesh = IntervalMesh(50, 0, 1)
Q = FunctionSpace(mesh, 'CG', 1)
B = FunctionSpace(mesh, 'B', 2)
M = Q + B

def left(x, on_boundary):
    return on_boundary and x[0] == 0

uD = project(Constant(0.0), M)
leftBC = DirichletBC(Q, 0.0, left)
leftBC_b = DirichletBC(Q, uD, left)

kappa = Constant(1.0/100.0)
s = Constant(5.0)
d = Constant(10.0)
f = Function(Q)
f.vector()[25] = 1000 # this is about the middle for a 50 element mesh

# =====
# standard Galerkin solution :
u = TrialFunction(Q)
v = TestFunction(Q)
us = Function(Q)
uf1 = Function(Q)

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + d * u.dx(0) * v * dx \
    + s * u * v * dx
L = f * v * dx

solve(a == L, us, leftBC)
uf1.interpolate(us)

# =====
# bubble-enriched solution :
u = TrialFunction(M)
v = TestFunction(M)
us = Function(M)
uf2 = Function(Q)

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + d * u.dx(0) * v * dx \
    + s * u * v * dx
L = f * v * dx

solve(a == L, us, leftBC_b)
uf2.interpolate(us)

# =====
# GLS stabilized solution :
u = TrialFunction(Q)
v = TestFunction(Q)
us = Function(Q)
uf3 = Function(Q)
h = CellSize(mesh)

# for SUPG :
# Pe = h * d / (2*kappa)
# tau = h / (2*d) * (1/tanh(Pe) - 1 / Pe)

# for GLS or SSM :
tau = 1 / (4*kappa/h**2 + 2*d/h + s)

def L(u):
    return -(kappa * u.dx(0)).dx(0) + d*u.dx(0) + s*u # GLS
def L_star(u):
    return -(kappa * u.dx(0)).dx(0) - d*u.dx(0) + s*u # SSM
def L_adv(u):
    return d*u.dx(0)

a = + kappa * u.dx(0) * v.dx(0) * dx \
    + d * u.dx(0) * v * dx \
    + s * u * v * dx \
    + inner(L(v), tau*L(u)) * dx
```

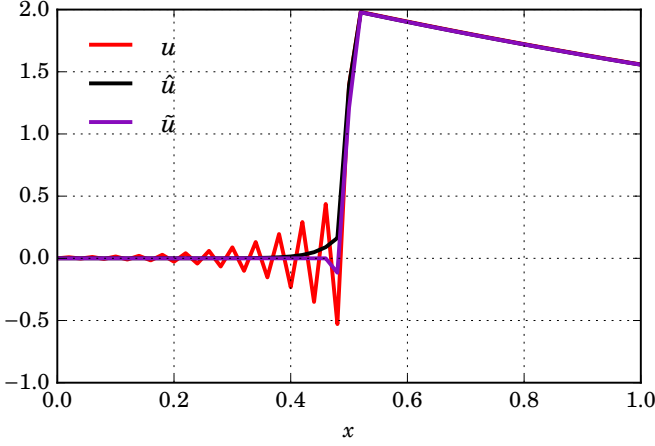



Figure 5.5: The unstabilized method solution u (red), bubble-enriched solution \hat{u} (black), and GLS-stabilized solution \tilde{u} (purple) with identical mesh spacing.

```

L = + f * v * dx \
    + inner(L(v), tau*f) * dx
solve(a == L, us, leftBC)
uf3.interpolate(us)

# =====
# plotting :
from pylab import *
purp = '#880c0c'

t = mesh.coordinates()[:,0][:-1]
uf1 = uf1.vector().array()
uf2 = uf2.vector().array()
uf3 = uf3.vector().array()

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['text.usetex'] = True
mpl.rcParams['text.latex.preamble'] = ['\usepackage{fouriernc}']

fig = figure(figsize=(5,3.5))
ax = fig.add_subplot(111)

ax.plot(t, uf1, 'r', ls='-', lw=2.0, label=r"$u$")
ax.plot(t, uf2, 'k', ls='-', lw=2.0, label=r"$\hat{u}$")
ax.plot(t, uf3, purp, ls='-', lw=2.0, label=r"$\tilde{u}$")

ax.set_xlabel(r'$x$')
ax.grid()
leg = ax.legend(loc='upper left')
leg.get_frame().set_alpha(0.0)
tight_layout()
savefig('../images/bubbles/extreme_new.pdf')

```

5.8 Stabilized Stokes equations

In this section we formulate a stabilized version of the slip-friction Stokes example described in §3.3 that circumvents inf-sup condition (3.15). Recall that the Stokes equations for incompressible fluid over the domain $\Omega = [0, 1] \times [0, 1]$ are

$$\mathcal{L}(\mathbf{u}, p) = -\nabla \cdot \sigma(\mathbf{u}, p) = \mathbf{f} \quad \text{in } \Omega \quad (5.26)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega, \quad (5.27)$$

where $\sigma(\mathbf{u}, p) = 2\eta \dot{\epsilon} - p\mathbf{I}$ is the Cauchy-stress tensor. The boundary conditions considered here are of type Dirichlet and traction (Neumann),

$$\mathbf{u} \cdot \mathbf{n} = g_D = 0 \quad \text{on } \Gamma_N, \Gamma_S, \Gamma_D \quad (5.28)$$

$$(\sigma \cdot \mathbf{n})_{\parallel} = \mathbf{g}_N = -\beta \mathbf{u} \quad \text{on } \Gamma_N, \Gamma_S, \Gamma_D \quad (5.29)$$

$$\mathbf{u} = \mathbf{g}_D = [-\sin(\pi y) \ 0]^\top \quad \text{on } \Gamma_E \quad (5.30)$$

$$\sigma \cdot \mathbf{n} = \mathbf{g}_N = [g_{N_x} \ g_{N_y}]^\top = \mathbf{0} \quad \text{on } \Gamma_W, \quad (5.31)$$

where $\Gamma_E, \Gamma_W, \Gamma_N$, and Γ_S are the East, West, North, and South boundaries, Γ_D is the dolphin boundary (Figure 5.6) and \mathbf{n} is the outward-pointing normal vector to these faces.

It has been shown by Hughes, Franca, and Balestra (1986) that the stabilized Galerkin approximate solution (\mathbf{u}, p) to Stokes system (5.26, 5.27) is given by solving the system

$$\begin{cases} (\Phi, \mathcal{L}(\mathbf{u}, p)) = (\Phi, \mathbf{f}), & \Phi \in \mathbf{S}_0^h, \\ (\xi, \nabla \cdot \mathbf{u}) + (\nabla \xi, \tau_S(\mathcal{L}(\mathbf{u}, p) - \mathbf{f})) = 0, & \xi \in M^h, \end{cases} \quad (5.32)$$

where the coefficient α in τ_S given by (5.23) is constrained to obey $0 < \alpha < \alpha_0$, where the upper bound α_0 depends on the basis used (the shape functions) for \mathbf{u} and Φ .

A modification was made to (5.32) by Hughes and Franca (1987) that allowed for discontinuous pressure spaces to be used. The form for this model is given by

$$B_{VII}(\mathbf{u}, p, \Phi, \xi) = L_{VII}(\Phi, \xi), \quad (5.33)$$

where

$$\begin{aligned} B_{VII}(\mathbf{u}, p, \Phi, \xi) = & +(\Phi, \mathcal{L}(\mathbf{u}, p)) - (\xi, \nabla \cdot \mathbf{u}) \\ & - (\mathcal{L}(\Phi, \xi), \tau_{S_\Omega} \mathcal{L}(\mathbf{u}, p)) - ([\xi], \tau_{S_{\Gamma'}} [p])_{\Gamma'}, \end{aligned} \quad (5.34)$$

$$L_{VII}(\Phi, \xi) = +(\Phi, \mathbf{f}) - (\mathcal{L}(\Phi, \xi), \tau_{S_\Omega} \mathbf{f}), \quad (5.35)$$

and

$$\tau_{S_\Omega} = \tau_S = \alpha \frac{h^2}{2\eta}, \quad \tau_{S_{\Gamma'}} = \zeta \frac{h^2}{2\eta}. \quad (5.36)$$

with constants $\alpha, \zeta \geq 0$ are dependent on the basis used for \mathbf{u} and Φ . Note that the notation $[\cdot]$ denotes jump across interior edges, i.e. across the $+$ and $-$ sides of an edge,

$$[\psi] = \psi^+ - \psi^-.$$

Therefore, if a continuous basis is used for p and ξ , ζ can be taken to be zero due to the fact that the jump terms in (5.33) will have no effect.

An independent analysis from Hughes and Franca (1987) was presented by Douglas and Wang (1989) possessing a remarkably similar form, but where α and ζ were shown to be shape-independent. This *absolutely stabilized* model possesses the form

$$B_{AS}(\mathbf{u}, p, \Phi, \xi) = L_{AS}(\Phi, \xi), \quad (5.37)$$

where

$$\begin{aligned} B_{AS}(\mathbf{u}, p, \Phi, \xi) = & +(\Phi, \mathcal{L}(\mathbf{u}, p)) + (\xi, \nabla \cdot \mathbf{u}) \\ & + (\mathcal{L}(\Phi, \xi), \tau_{S_\Omega} \mathcal{L}(\mathbf{u}, p)) + ([\xi], \tau_{S_{\Gamma'}} [p])_{\Gamma'}, \end{aligned} \quad (5.38)$$

$$L_{AS}(\Phi, \xi) = +(\Phi, \mathbf{f}) + (\mathcal{L}(\Phi, \xi), \tau_{S_\Omega} \mathbf{f}), \quad (5.39)$$

and utilizes the same coefficients τ_{S_Ω} and $\tau_{S_{\Gamma'}}$ defined by (5.36) with the difference that they possess only the single positivity constraint $\alpha, \zeta \geq 0$. Note that the only difference between

(5.33) and (5.37) is the sign of the last two terms of bilinear forms (5.34, 5.38) and the last term of linear forms (5.35, 5.39).

The Galerkin/least-squares stabilized bilinear form for Dirichlet-traction-Stokes system (5.26, 5.27, 5.28 – 5.31) are found identically to the formation of Nitsche variational form (3.20); integration by parts of $(\Phi, \mathcal{L}(\mathbf{u}, p))$ and the addition of symmetric Nitsche terms, with the incorporation of the extra GLS terms of (5.37),

$$\mathcal{B}_\Omega + \mathcal{B}_{\Gamma_G} + \mathcal{B}_{\Gamma_G}^W + \mathcal{B}_{\Gamma_E} + \mathcal{B}_{\Gamma_E}^W + \mathcal{B}_\Omega^B + \mathcal{B}_{\Gamma'}^B = \mathcal{F} + \mathcal{F}^W + \mathcal{F}^B, \quad (5.40)$$

with individual terms

$$\begin{aligned} \mathcal{B}_\Omega &= + \int_\Omega \sigma(\mathbf{u}, p) : \nabla \Phi \, d\Omega + \int_\Omega (\nabla \cdot \mathbf{u}) \xi \, d\Omega \\ \mathcal{B}_{\Gamma_G} &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\mathbf{u}, p) \cdot \mathbf{n}) \mathbf{n} \cdot \Phi \, d\Gamma_G + \int_{\Gamma_G} \beta \mathbf{u} \cdot \Phi \, d\Gamma_G \\ \mathcal{B}_{\Gamma_G}^W &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) \mathbf{n} \cdot \mathbf{u} \, d\Gamma_G + \gamma \int_{\Gamma_G} \frac{1}{h} (\mathbf{u} \cdot \mathbf{n}) (\Phi \cdot \mathbf{n}) \, d\Gamma_G \\ \mathcal{B}_{\Gamma_E} &= - \int_{\Gamma_E} \sigma(\mathbf{u}, p) \cdot \mathbf{n} \cdot \Phi \, d\Gamma_E \\ \mathcal{B}_{\Gamma_E}^W &= - \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{u} \, d\Gamma_E + \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{u}) \, d\Gamma_E \\ \mathcal{B}_\Omega^B &= + \frac{\alpha}{2} \int_\Omega \frac{h^2}{\eta} \mathcal{L}(\Phi, \xi) \mathcal{L}(\mathbf{u}, p) \, d\Omega \\ \mathcal{B}_{\Gamma'}^B &= + \frac{\zeta}{2} \int_{\Gamma'} \frac{h^2}{\eta} [\xi][p] \, d\Gamma' \end{aligned}$$

and

$$\begin{aligned} \mathcal{F} &= + \int_\Omega \mathbf{f} \cdot \Phi \, d\Omega \\ \mathcal{F}^W &= - \int_{\Gamma_G} (\mathbf{n} \cdot \sigma(\Phi, \xi) \cdot \mathbf{n}) g_D \, d\Gamma_G + \gamma \int_{\Gamma_G} \frac{1}{h} g_D \Phi \cdot \mathbf{n} \, d\Gamma_G \\ &\quad - \int_{\Gamma_E} \sigma(\Phi, \xi) \cdot \mathbf{n} \cdot \mathbf{g}_D \, d\Gamma_E + \gamma \int_{\Gamma_E} \frac{1}{h} (\Phi \cdot \mathbf{g}_D) \, d\Gamma_E \\ \mathcal{F}^B &= + \frac{\alpha}{2} \int_\Omega \frac{h^2}{\eta} \mathcal{L}(\Phi, \xi) f \, d\Omega \end{aligned}$$

where $\Gamma_G = \Gamma_N \cup \Gamma_S \cup \Gamma_D$ is the entire slip-friction boundary, h is the element diameter, and $\gamma > 0$ is an application-specific parameter normally derived by experimentation (see §3.3).

The mixed variational formulation consistent with problem (5.26, 5.27, 5.28 – 5.31) reads: find mixed approximation $\mathbf{u}, p \in (\mathbf{S}_E^h \subset (\mathcal{H}_E^1(\Omega))^2) \times (M^h \subset L^2(\Omega))$ subject to (5.40) for all $\Phi, \xi \in (\mathbf{S}_0^h \subset (\mathcal{H}_{E_0}^1(\Omega))^2) \times (M^h \subset L^2(\Omega))$.

The velocity and pressure solutions to this problem using linear Lagrange elements for both \mathbf{u} and p are depicted in Figure 5.6, and were generated by Code Listing 5.5.

Code Listing 5.5: FEniCS solution to the 2D-Nitsche-stabilized-Stokes-slip-friction problem of §5.8.

```
from fenics import *
mesh = Mesh("meshes/dolphin_fine.xml.gz")
sub_domains = MeshFunction("size_t", mesh,
                           "meshes/dolphin_fine_subdomains.xml.gz")
```

```
# P1 - P1 mixed element :
V = VectorFunctionSpace(mesh, 'CG', 1)
Q = FunctionSpace(mesh, 'CG', 1)
W = V*Q

# variational problem
u, p = TrialFunctions(W)
v, q = TestFunctions(W)

# no penetration boundary condition for velocity
# y = 0, y = 1 and around the dolphin :
u_n = Constant(0.0)

# Inflow boundary condition for velocity at x = 1 :
u_0 = Expression(("sin(x[1]*pi)", "0.0"))

# relevant measures :
ds = Measure("ds")[sub_domains]
dG_0 = ds(0)
dG_r = ds(1)

# constants :
alpha = Constant(1.0/10.0)
gamma = Constant(1e2)
h = CellSize(mesh)
n = FacetNormal(mesh)
I = Identity(2)
eta = Constant(1.0)
f = Constant((0.0, 0.0))
beta = Constant(10.0)

def epsilon(u): return 0.5*(grad(u) + grad(u).T)
def sigma(u, p): return 2*eta * epsilon(u) - p*I
def L(u, p): return -div(sigma(u, p))

t = dot(sigma(u, p), n)
s = dot(sigma(v, q), n)

B_o = + inner(sigma(u, p), grad(v)) * dx + div(u) * q * dx \
      + alpha * h**2 * inner(L(u, p), L(v, q)) * dx

B_g = - dot(n, t) * dot(v, n) * dG_0 \
      - dot(u, n) * dot(s, n) * dG_0 \
      + gamma/h * dot(u, n) * dot(v, n) * dG_0 \
      + beta * dot(u, v) * dG_0 \
      - inner(dot(sigma(u, p), n), v) * dG_r \
      - inner(dot(sigma(v, q), n), u) * dG_r \
      + gamma/h * inner(v, u) * dG_r

F = + dot(f, v) * dx \
      + gamma/h * u_n * dot(v, n) * dG_0 \
      - inner(dot(sigma(v, q), n), u_0) * dG_r \
      + gamma/h * inner(v, u_0) * dG_r \
      + alpha * h**2 * inner(f, L(v, q)) * dx

# solve variational problem
wh = Function(W)
solve(B_o + B_g == F, wh)
uh, ph = wh.split(True)

print "Norm of velocity coefficient vector: %.15g" % uh.vector().norm("l2")
print "Norm of pressure coefficient vector: %.15g" % ph.vector().norm("l2")

# get individual components with deep copy :
u0, u1 = uh.split(True)

from pylab import *
from mpl_toolkits.axes_grid1 import make_axes_locatable
from matplotlib import colors, ticker

# calculate array components :
v0 = u0.compute_vertex_values(mesh)
v1 = u1.compute_vertex_values(mesh)
v = sqrt(v0**2 + v1**2 + 1e-16)
v0 = v0 / v
v1 = v1 / v
x = mesh.coordinates()[0, :]
y = mesh.coordinates()[1, :]
t = mesh.cells()

# generate velocity figure :
fig = figure(figsize=(8, 7))
ax = fig.add_subplot(111)

v[v > 2.0] = 2.0
cm = get_cmap('viridis')
ls = array([0.0, 0.1, 0.2, 0.4, 0.6, 0.8, 1.0, 1.2, 1.4, 1.6, 1.8, 2.00001])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-', color='k', lw=0.2, alpha=0.3)
q = ax.quiver(x, y, v0, v1, pivot='middle',
              color='k',
              scale=60,
              width=0.0015,
              headwidth=4.0,
              headlength=4.0,
              headaxislength=4.0)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])

divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = fig.colorbar(c, cax=cax, ticks=ls, format='%1.1f')
tight_layout()
savefig('../images/fenics_intro/2DStokes_nitsche_u_stab.pdf')

# generate pressure figure :
v = ph.compute_vertex_values(mesh)

fig = figure(figsize=(8, 7))
ax = fig.add_subplot(111)

v[v > 120] = 120
v[v < -20] = -20

ls = array([v.min(), -10, 0, 10, 20, 30, 40, 50, 60, 70, 80, 90, 100, 110, 120])
nm = colors.BoundaryNorm(ls, cm.N)
c = ax.tricontourf(x, y, t, v, 10, cmap=cm, norm=nm, levels=ls)
tp = ax.triplot(x, y, t, '-', color='k', lw=0.2, alpha=0.5)

ax.set_xlabel(r'$x$')
ax.set_ylabel(r'$y$')
ax.axis('equal')
ax.set_xlim([x.min(), x.max()])
ax.set_ylim([y.min(), y.max()])
ax.set_xticklabels([])
ax.set_yticklabels([])
```



```
divider = make_axes_locatable(gca())
cax = divider.append_axes('right', "5%", pad="3%")
cbar = colorbar(c, cax=cax)
tight_layout()
savefig('.../images/fenics_intro/2Dstokes_nitsche_p_stab.pdf')
```

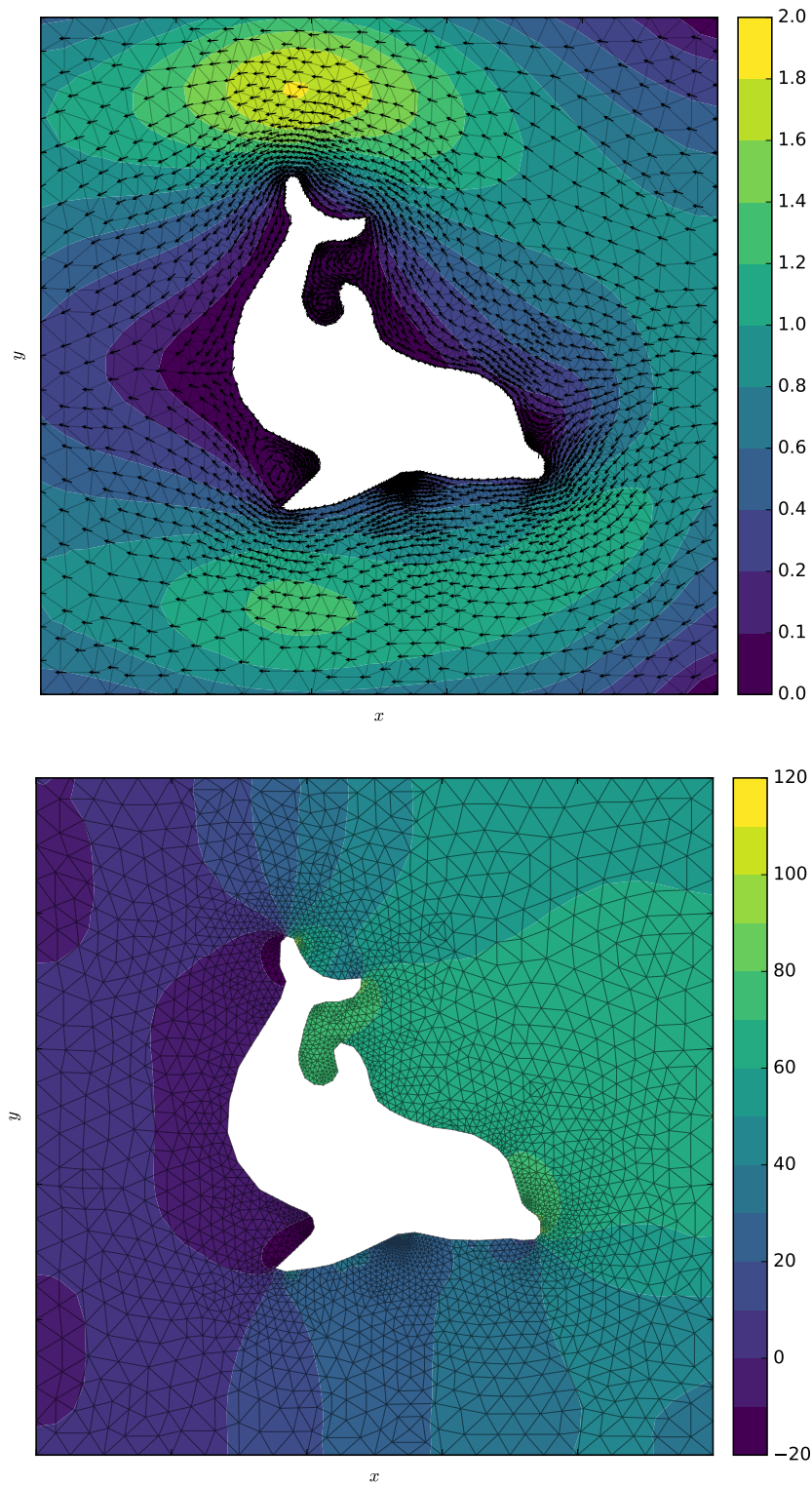


Figure 5.6: Galerkin/least-squares stabilized velocity field \mathbf{u} (top) and pressure p (bottom) with $\alpha = 0.1$, $\zeta = 0$, $\beta = 10$, and $\gamma = 100$, utilizing continuous linear Lagrange elements for both \mathbf{u} and p (referred to as P1 – P1 approximation).

Chapter 6

Nonlinear solution process

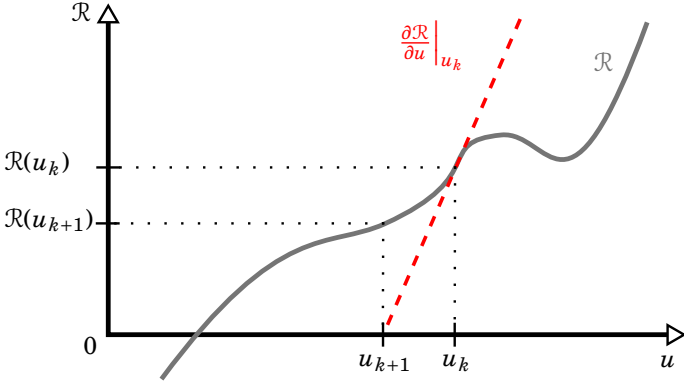


Figure 6.1: Illustration of the Newton-Raphson method for solving (6.1) for u .

All of the examples presented thus far have been linear equations. For non-linear systems, the solution method described in §1.2.6 no longer apply. For these problems the n unknown degrees of freedom of u – with vector representation \mathbf{u} – may be uniquely determined by solving for the down gradient direction of a quadratic model of the functional *residual*

$$\mathcal{R}(u) = 0, \quad (6.1)$$

formed by moving all terms of a variational form to one side of the equation. The following sections explain two ways of solving problems of this nature.

6.1 Newton-Raphson method

One way to solve system (6.1) is the *Newton-Raphson* method (Nocedal and Wright, 2000). This method effectively linearizes the problem by first assuming an initial guess of the minimizer, \mathbf{u}_k , and the functional desired to be minimized, $\mathcal{R}(\mathbf{u}_k) = \mathcal{R}_k$, then uses the \mathcal{R} -intercept of the tangent line to this guess as a subsequent guess, \mathbf{u}_{k+1} . This procedure is repeated until either the absolute value of \mathcal{R} is below a desired *absolute tolerance* or the relative change of \mathcal{R} between guesses \mathbf{u}_k and \mathbf{u}_{k+1} is below a desired *relative tolerance* (Figure 6.1).

6.1.1 Procedure

First, as elaborated upon by Nocedal and Wright (2000), the Taylor-series approximation of the residual $\mathcal{R}(\mathbf{u}_k)$ perturbed in a direction \mathbf{p}_k provides a quadratic model $m_k(\mathbf{p}_k)$,

$$\mathcal{R}(\mathbf{u}_k + \mathbf{p}_k) \approx \mathcal{R}_k + \mathbf{p}_k^\top \nabla \mathcal{R}_k + \frac{1}{2} \mathbf{p}_k^\top \nabla^2 \mathcal{R}_k \mathbf{p}_k =: m_k(\mathbf{p}_k). \quad (6.2)$$

Because we wish to find the search direction \mathbf{p}_k which minimizes $m_k(\mathbf{p}_k)$, we set the gradient of this function equal to zero and solve

$$\begin{aligned} 0 &= \nabla m_k(\mathbf{p}_k) \\ &= \nabla \mathcal{R}_k + \nabla (\mathbf{p}_k^\top \nabla \mathcal{R}_k) + \nabla \left(\frac{1}{2} \mathbf{p}_k^\top \nabla^2 \mathcal{R}_k \mathbf{p}_k \right) \\ &= \nabla \mathcal{R}_k + \nabla^2 \mathcal{R}_k \mathbf{p}_k, \end{aligned} \quad (6.3)$$

giving us the *Newton direction*

$$\mathbf{p}_k = -(\nabla^2 \mathcal{R}_k)^{-1} \nabla \mathcal{R}_k. \quad (6.4)$$

Quadratic model (6.2) is simplified to a linear model if the second derivative is eliminated. Integrating (6.3) over \mathbf{u}_k produces

$$\begin{aligned} \nabla^2 \mathcal{R}_k \mathbf{p}_k &= -\nabla \mathcal{R}_k \\ \int_{\mathbf{u}_k} \nabla^2 \mathcal{R}_k(\mathbf{u}_k) \mathbf{p}_k d\mathbf{u}_k &= - \int_{\mathbf{u}_k} \nabla \mathcal{R}_k(\mathbf{u}_k) d\mathbf{u}_k \\ \nabla \mathcal{R}_k \mathbf{p}_k &= -\mathcal{R}_k, \end{aligned} \quad (6.5)$$

a discrete system of equations representing a first-order differential equation for Newton direction \mathbf{p}_k . To complete system (6.5), one boundary condition may be specified. For all Dirichlet boundaries, u is known and can therefore not be improved. Hence search direction $p = 0$ over essential boundaries. Boundaries corresponding to natural conditions of u require no specific treatment.

The iteration with *step length* or *relaxation parameter* $\alpha \in (0, 1]$ is defined as

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha \mathbf{p}_k. \quad (6.6)$$

Provided that the curve \mathcal{R} is relatively smooth and that the Hessian matrix $\nabla^2 \mathcal{R}(\mathbf{u}_k)$ in (6.4) is positive definite, (6.6) describes an iterative procedure for calculating the minimum of \mathcal{R} and corresponding optimal value of \mathbf{u} . See Algorithm 1 and CSLVR source code 6.1 for details.

6.1.2 Gâteaux derivatives

Because residual (6.1) is a functional, *Gâteaux derivatives* are used to calculate the directional derivatives in (6.5). To illustrate this derivative, consider the second-order boundary-value problem

$$\begin{aligned} \nabla^2 u - u &= 0 & \text{in } \Omega \\ \nabla u \cdot \mathbf{n} &= g & \text{on } \Gamma. \end{aligned}$$

where Ω is the interior domain with boundary Γ , and \mathbf{n} is the outward-pointing normal vector. The associated weak form is

$$\mathcal{R}(u) = - \int_{\Omega} \nabla u \cdot \nabla \phi d\Omega - \int_{\Omega} \phi u d\Omega + \int_{\Gamma} \phi g d\Gamma = 0, \quad (6.7)$$

The Gâteaux derivative or first variation of \mathcal{R} with respect to u in the direction p – with vector notation $\delta_{\mathbf{u}}^p \mathcal{R}(\mathbf{u})$ – is defined as

$$\begin{aligned} \delta_{\mathbf{u}}^p \mathcal{R}(u) &= \frac{\delta}{\delta u} \mathcal{R}(u, p) = \lim_{\epsilon \rightarrow 0} \frac{d}{d\epsilon} \mathcal{R}(u + \epsilon p) \\ &= - \frac{d}{d\epsilon} \left[\int_{\Omega} \nabla(u + \epsilon p) \cdot \nabla \phi d\Omega \right]_{\epsilon=0} \\ &\quad - \frac{d}{d\epsilon} \left[\int_{\Omega} \phi(u + \epsilon p) d\Omega \right]_{\epsilon=0} \\ &\quad + \frac{d}{d\epsilon} \left[\int_{\Gamma} \phi g d\Gamma \right]_{\epsilon=0} \\ &= - \int_{\Omega} \nabla p \cdot \nabla \phi d\Omega - \int_{\Omega} \phi p d\Omega. \end{aligned}$$

It is important to recognize that if p is a function with known values, i.e. data interpolated onto a finite-element mesh, the finite-element assembly – described in §1.2.4 – will result in a vector of length n . However, if p is an unknown quantity and thus a member of the trial space associated with the finite-element approximation of \mathbf{u} , as is the case with the Newton-Raphson method here, the assembly process will result in a matrix with properties identical to the associated stiffness matrix of (6.7). In order to clearly differentiate between these circumstances, the Gâteaux derivative operator notation $\delta_{\mathbf{u}}^p$ is used when \mathbf{p} is a member of the trial space, and $\delta_{\mathbf{u}}$ otherwise.

These derivatives may be calculated with FEniCS using the process of *automatic differentiation* (Nocedal and Wright, 2000), as illustrated by the nonlinear problem example in Code Listing 6.2.

Code Listing 6.1: Newton-Raphson method as implemented in CSLVR's model class

Algorithm 1 - Newton-Raphson method

```

1: INPUTS:
2:    $\mathcal{R}$  - residual variational form
3:    $\mathbf{u}$  - initial state parameter vector
4:    $\hat{\mathbf{p}}$  - trial function in same space as  $\mathbf{u}$ 
5:    $\alpha$  - relaxation parameter
6:    $a_{tol}$  - absolute tolerance to stop iterating
7:    $r_{tol}$  - relative tolerance to stop iterating
8:    $n_{max}$  - maximum iterations
9: OUTPUT:
10:   $\mathbf{u}^*$  - optimized state parameter vector
11:
12: function NR( $\mathcal{R}, \mathbf{u}, \hat{\mathbf{p}}, \alpha, a_{tol}, r_{tol}, n_{max}$ )
13:    $r := \infty$ 
14:    $a := \infty$ 
15:   while ( $a > a_{tol}$  or  $r > r_{tol}$ ) and  $n < n_{max}$  do
16:      $J := \text{assemble } \delta_{\mathbf{u}}^{\hat{\mathbf{p}}} \mathcal{R}(\mathbf{u})$ 
17:      $\mathbf{b} := \text{assemble } \mathcal{R}(\mathbf{u})$ 
18:      $\mathbf{p} \leftarrow \text{solve } J\mathbf{p} = -\mathbf{b}$ 
19:      $\mathbf{u} := \mathbf{u} + \alpha\mathbf{p}$ 
20:      $a := \|\mathbf{b}\|_2$ 
21:     if  $n = 0$  then
22:        $a_0 := a$ 
23:     end if
24:      $r := a/a_0$ 
25:      $n := n + 1$ 
26:   end while
27:   return  $\mathbf{u}^* := \mathbf{u}$ 
28: end function

```

```

def home_rolled_newton_method(self, R, U, J, bcs, atol=1e-7, rtol=1e-10,
                             relaxation_param=1.0, max_iter=25,
                             method='cg', preconditioner='amg',
                             cb_ftn=None):
    """
    Apply Newton's method.
    Args:
        :R: residual of system
        :U: unknown to determine
        :J: Jacobian
        :bcs: set of Dirichlet boundary conditions
        :atol: absolute stopping tolerance
        :rtol: relative stopping tolerance
        :relaxation_param: ratio of down-gradient step to take each iteration.
        :max_iter: maximum number of iterations to perform
        :method: linear solution method
        :preconditioner: preconditioning method to use with 'Krylov' solver
        :cb_ftn: at the end of each iteration, this is called
    """
    converged = False

```

```

lambda      = relaxation_param    # relaxation parameter
nIter       = 0                  # number of iterations

# need to homogenize the boundary, as the residual is always zero over
# essential boundaries :
bcs_u = []
for bc in bcs:
    bc = DirichletBC(bc)
    bc.homogenize()
    bcs_u.append(bc)

# the direction of decent :
d = Function(U.function_space())

while not converged and nIter < max_iter:
    # assemble system :
    A, b = assemble_system(J, -R, bcs_u)

    # determine step direction :
    solve(A, d.vector(), b, method, preconditioner, annotate=False)

    # calculate residual :
    residual = b.norm('l2')

    # set initial residual :
    if nIter == 0:
        residual_0 = residual

    # the relative residual :
    rel_res = residual/residual_0

    # check for convergence :
    converged = residual < atol or rel_res < rtol

    # move U down the gradient :
    U.vector()[1:] += lambda*d.vector()

    # increment counter :
    nIter += 1

# print info to screen :
if self.MPI_rank == 0:
    string = "Newton iteration %d: r (abs) = %.3e (tol = %.3e) " \
            + "r (rel) = %.3e (tol = %.3e)"
    print string % (nIter, residual, atol, rel_res, rtol)

# call the callback function, if desired :
if cb_ftn is not None:
    s = " ::: calling home-rolled Newton method callback ::: "
    print_text(s, cls=self.this)
    cb_ftn()

```

6.2 Nonlinear problem example

Suppose we would like to minimize the time for a boat to cross a river. The time for this boat to cross, when steered directly perpendicular to the river's parallel banks is given by

$$\begin{aligned}
 T(y) &= \int_0^T dt = \int_0^S \frac{dt}{ds} ds = \int_0^S \frac{1}{\|\mathbf{u}\|} ds \\
 T(y) &= \int_0^\ell \frac{\sqrt{1+(y')^2}}{\|\mathbf{u}\|} dx \\
 T(y) &= \int_0^\ell \frac{\sqrt{1+(y')^2}}{\sqrt{v^2+u^2}} dx = \int_0^\ell L(x, y') dx,
 \end{aligned}$$

where S is the length of the boat's path; \mathbf{u} is the velocity of the boat with components river current speed $v(x)$ and boat speed u in the x direction as a result of its motor; and ℓ is the width of the river. For *Lagrangian* L , the first variation of T in the direction of the test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ (see test space (1.11)) is

$$\begin{aligned}
 \delta T(y, \phi) &= \frac{d}{d\epsilon} \int_0^\ell L(x, y + \epsilon\phi, y' + \epsilon\phi') dx \Big|_{\epsilon=0} \\
 &= \int_0^\ell (L_y \phi + L_{y'} \phi') dx \\
 &= \int_0^\ell \left(L_y - \frac{d}{dx} L_{y'} \right) \phi dx + L_{y'} \phi \Big|_{x=0}^{x=\ell} \\
 &= \int_0^\ell (L_y - [L_{y'x} + L_{y'y'} y' + L_{y'y'} y'']) \phi dx + L_{y'} \phi \Big|_{x=0}^{x=\ell}.
 \end{aligned}$$

Evaluating each individual term,

$$\begin{aligned}
 L_y &= 0 \\
 L_{y'}(y') &= \frac{y' (1+(y')^2)^{-1/2}}{\sqrt{v^2+u^2}} \\
 L_{y'x}(y') &= \frac{(2(y')^2+1)y''}{\sqrt{u^2+v^2}\sqrt{1+(y')^2}} \\
 L_{y'y} &= 0 \\
 L_{y'y'}(y') &= \frac{1}{\sqrt{u^2+v^2}\sqrt{1+(y')^2}} \left(1 - \frac{(y')^2}{(1+(y')^2)^3} \right),
 \end{aligned}$$

and so the first variation of T is reduced to

$$\delta T(y, \phi) = - \int_0^\ell (L_{y'x} + L_{y'y'} y'') \phi dx + L_{y'} \phi \Big|_{x=0}^{x=\ell}. \quad (6.8)$$

In order to find the minimal time for the boat to cross, this first variation of T must be equal to zero. First, defining

$$M = \frac{L_{y'x}}{y''},$$

relation (6.8) can be rewritten followed by integrating by parts of the the second-derivative term once again,

$$\begin{aligned}
 0 &= - \int_0^\ell (M y'' + L_{y'y'} y'') \phi dx + L_{y'} \phi \Big|_{x=0}^{x=\ell} \\
 &= - \int_0^\ell (M + L_{y'y'}) y'' \phi dx + L_{y'} \phi \Big|_{x=0}^{x=\ell} \\
 &= \int_0^\ell (M + L_{y'y'}) y' \phi' dx - [M + L_{y'y'}] y' \phi \Big|_{x=0}^{x=\ell} + L_{y'} \phi \Big|_{x=0}^{x=\ell}.
 \end{aligned}$$

If the left essential boundary condition is set to $y(0) = 0$, and the right natural boundary condition is set to be equal to the trajectory of the boat at the opposite bank given by

$$y'(\ell) = g = \frac{v(\ell)}{u},$$

the final variational problem therefore consists of finding $y \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space (1.10)) such that

$$0 = \int_0^\ell (M(y') + L_{y'y'}(y')) y' \phi' dx - [M(g) + L_{y'y'}(g)] g \phi \Big|_{x=\ell} + L_{y'}(g) \phi \Big|_{x=\ell}.$$

for all $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$.

The weak solution to this equation using linear-Lagrange shape functions (1.18) using the built-in FEniCS Newton solver is shown in Figure 6.2 and generated from Code Listing 6.2.

Code Listing 6.2: FEniCS source code for the river crossing example.

```

from fenics import *

mesh = IntervalMesh(1000, 0, 10)
Q = FunctionSpace(mesh, 'CG', 1)

v = interpolate(Expression('10*exp(-0.5*pow(x[0] - 5.0, 2)/pow(2,2))'), Q)
u = 1.0
g = v / u

y = Function(Q)
dy = TrialFunction(Q)
phi = TestFunction(Q)

def left(x, on_boundary):

```

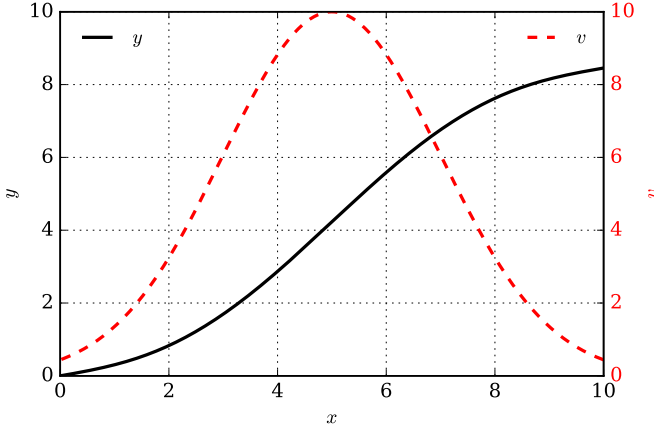


Figure 6.2: Path taken for a boat to cross a river, $y(x)$ (solid black), with a motor speed in the x -direction of $u = 1$, and river velocity in the y -direction $v(x)$ (dashed red).

```

return on_boundary and abs(x[0]) < 1e-14

# boat starts at y=0 :
bc = DirichletBC(Q, 0.0, left)

def Lp(yp):
    Lp = yp * (1 + yp**2)**(-1/2.) / sqrt(v**2 + u**2)
    return Lp

def Lpp(yp):
    Lpp = 1/(sqrt(1 + yp**2) * sqrt(v**2 + u**2)) * (1 - yp**2 / (1 + yp**2)**3)
    return Lpp

def M(yp):
    q = (2*yp**2 + 1) / (sqrt(u**2 + v**2) * sqrt(yp**2 + 1))
    return q

F = + (Lpp(y.dx(0)) + M(y.dx(0))) * y.dx(0) * phi.dx(0) * dx \
    - (M(g) + Lpp(g)) * g * phi * ds \
    + Lp(g) * phi * ds

J = derivative(F, y, dy)

solve(F == 0, y, bc, J=J)

# =====
# plot :

from pylab import *

mpl.rcParams['font.family'] = 'serif'
mpl.rcParams['legend.fontsize'] = 'medium'

x = mesh.coordinates()[0,:][:-1]
vf = v.vector().array()
yf = y.vector().array()

fig = figure(figsize=(6,4))
ax1 = fig.add_subplot(111)
ax2 = ax1.twinx()

ax1.plot(x, yf, 'k', lw=2.0, label=r"$y$")
ax2.plot(x, vf, 'r--', lw=2.0, label=r"$v$")

ax2.tick_params(axis='x', colors='r')
ax2.yaxis.label.set_color('r')
ax2.set_ylabel(r'$v$')
leg2 = ax2.legend(loc='upper right')
leg2.get_frame().set_alpha(0.0)

for tl in ax2.get_yticklabels():
    tl.set_color('r')

#ax1.set_ylim(0, 2*pi)
ax1.set_ylim(0, 10)
ax1.set_xlabel(r'$x$')
ax1.set_ylabel(r'$y$')
ax1.grid()
leg1 = ax1.legend(loc='upper left')
leg1.get_frame().set_alpha(0.0)

tight_layout()
savefig("../images/fenics_intro/river_cross.pdf")
show()

```

6.3 Quasi-Newton solution process

In some situations, full quadratic model (6.2) may be preferred over linear model (6.5) for solving a non-linear system. When the inverse of Hessian matrix $\nabla^2 \mathcal{R}_k$ is difficult to calculate by hand, it may be approximated by using curvature information

at a current guess \mathbf{u}_k . Algorithms utilizing the Hessian approximation are referred to as *quasi-Newton* methods.

As described by Nocedal and Wright (2000), the most modern and efficient of such Hessian approximation techniques is that proposed by Broyden, Fletcher, Goldfarb, and Shanno, aptly referred to as the BFGS method (Algorithm 2). This method uses the iteration

$$\mathbf{u}_{k+1} = \mathbf{u}_k + \alpha_k \mathbf{p}_k \quad (6.9)$$

and a quadratic model similar to (6.2) with the addition of a subsequent iteration $k+1$ quadratic model:

$$m_k(\mathbf{p}) = \mathcal{R}_{k+1} + \mathbf{p}^\top \nabla \mathcal{R}_k + \frac{1}{2} \mathbf{p}^\top B_k \mathbf{p} \quad (6.10)$$

$$m_{k+1}(\mathbf{p}) = \mathcal{R}_{k+1} + \mathbf{p}^\top \nabla \mathcal{R}_{k+1} + \frac{1}{2} \mathbf{p}^\top B_{k+1} \mathbf{p}. \quad (6.11)$$

The minimizer of (6.10), search direction \mathbf{p}_k , is found identically to Newton direction (6.4):

$$\mathbf{p}_k = -B_k^{-1} \nabla \mathcal{R}_k,$$

where the approximate Hessian matrix $B_k \approx \nabla_{\mathbf{u}\mathbf{u}}^2 \mathcal{R}_k$ must be symmetric and positive definite. Furthermore, it is required that $\nabla m_{k+1} = \nabla \mathcal{R}_j$ for $j = k, k+1$, the last two iterates. For the last iterate $k+1$, \mathcal{R}_{k+1} is evaluated at \mathbf{u}_{k+1} and therefore the gradient of m_{k+1} at $\mathbf{p} = \mathbf{0}$ is evaluated,

$$\nabla m_{k+1}(\mathbf{0}) = \nabla \mathcal{R}_{k+1},$$

implying that the second condition is satisfied automatically. Using the same reasoning and iteration (6.9), to evaluate m_{k+1} at \mathbf{u}_k the gradient of m_{k+1} at $\mathbf{p} = -\alpha_k \mathbf{p}_k$ is evaluated,

$$\nabla m_{k+1}(-\alpha_k \mathbf{p}_k) = \nabla \mathcal{R}_{k+1} - \alpha_k \mathbf{p}_k B_{k+1} = \nabla \mathcal{R}_k,$$

thus requiring that

$$\alpha_k \mathbf{p}_k B_{k+1} = \nabla \mathcal{R}_{k+1} - \nabla \mathcal{R}_k.$$

Using iteration (6.9), the *secant equation* is

$$B_{k+1} \mathbf{s}_k = \mathbf{y}_k, \quad (6.12)$$

where

$$\mathbf{s}_k = \mathbf{u}_{k+1} - \mathbf{u}_k \quad \mathbf{y}_k = \nabla \mathcal{R}_{k+1} - \nabla \mathcal{R}_k.$$

Equivalently, the *inverse secant equation* is

$$H_{k+1} \mathbf{y}_k = \mathbf{s}_k, \quad (6.13)$$

where $H_{k+1} = B_{k+1}^{-1}$.

As described by Nocedal and Wright (2000), inverse secant equation (6.13) will be satisfied if the *curvature condition*

$$\mathbf{s}_k^\top \mathbf{y}_k > 0$$

holds. This is explicitly enforced by choosing step length α_k in (6.9) such that the *Armijo condition*

$$\mathcal{R}(\mathbf{u}_k + \alpha_k \mathbf{p}_k) \leq \mathcal{R}_k + c_1 \alpha_k \mathbf{p}_k^\top \nabla \mathcal{R}_k \quad (6.14)$$

and the curvature condition

$$\mathbf{p}^\top \nabla \mathcal{R}(\mathbf{u}_k + \alpha_k \mathbf{p}_k) \geq c_2 \mathbf{p}^\top \nabla \mathcal{R}_k, \quad (6.15)$$

collectively referred to as the *Wolfe conditions*. *Wolfe conditions*, hold for some pair of constants $c_1, c_2 \in (0, 1)$. One way of enforcing (6.14) and (6.15) is described by the *backtracking line search*, Algorithm 3 (Nocedal and Wright, 2000).

In order to derive a unique H_{k+1} , the additional constraint that H_{k+1} be close to the current matrix H_k is imposed. Thus H_{k+1} is the solution to the problem

$$\min_H \|H - H_k\| \quad (6.16)$$

$$\text{subject to } H = H^\top, \quad H \mathbf{y}_k = \mathbf{s}_k. \quad (6.17)$$

Using the weighted Frobenius norm

$$\|A\|_W = \left\| W^{1/2} A W^{1/2} \right\|_F$$

with average Hessian inverse

$$W = \left[\int_0^1 \nabla^2 \mathcal{R}(\mathbf{u} + \tau \alpha_k \mathbf{p}_k) d\tau \right]^{-1}$$

satisfying $W \mathbf{y}_k = \mathbf{s}_k$ in (6.16) gives the unique solution to (6.16) and (6.17)

$$H_{k+1} = (I - \rho_k \mathbf{s}_k \mathbf{y}_k^\top) H_k (I - \rho_k \mathbf{y}_k \mathbf{s}_k^\top) + \rho_k \mathbf{s}_k \mathbf{s}_k^\top, \quad \rho = (\mathbf{y}^\top \mathbf{s})^{-1}.$$

The iterative process for this method is described by Algorithm 2.

Algorithm 2 - BFGS quasi-Newton method

```

1: INPUTS:
2:    $\mathbf{u}$  - initial state parameter vector
3:    $H$  - inverse Hessian approximation
4:    $a_{tol}$  - absolute tolerance to stop iterating
5:    $r_{tol}$  - relative tolerance to stop iterating
6: OUTPUT:
7:    $\mathbf{u}^*$  - optimized state parameter vector
8:
9: function BFGS( $\mathbf{u}, H, a_{tol}$ )
10:    $r := \infty$ 
11:    $a := \infty$ 
12:   while  $a > a_{tol}$  or  $r > r_{tol}$  do
13:      $\mathbf{g} := \text{assemble } \delta_{\mathbf{u}} \mathcal{R}(\mathbf{u})$ 
14:      $\mathbf{p} := -H \mathbf{g}$ 
15:      $\alpha := \text{BLS}(\mathbf{p}, \mathbf{g}, \mathbf{u})$ 
16:      $\mathbf{u}_k := \mathbf{u} + \alpha \mathbf{p}$ 
17:      $\mathbf{g}_k := \text{assemble } \delta_{\mathbf{u}} \mathcal{R}(\mathbf{u}_k)$ 
18:      $\mathbf{s} := \mathbf{u}_k - \mathbf{u}$ 
19:      $\mathbf{y} := \mathbf{g}_k - \mathbf{g}$ 
20:      $\rho := (\mathbf{y}^\top \mathbf{s})^{-1}$ 
21:      $H := (I - \rho \mathbf{s} \mathbf{y}^\top) H (I - \rho \mathbf{y} \mathbf{s}^\top) + \rho \mathbf{s} \mathbf{s}^\top$ 
22:      $a := \|\mathbf{g}_k\|_\infty$ 
23:      $r := \|\mathbf{u} - \mathbf{u}_k\|_\infty$ 
24:      $\mathbf{u} := \mathbf{u}_k$ 
25:   end while
26:   return  $\mathbf{u}^* := \mathbf{u}$ 
27: end function

```

Algorithm 3 - Backtracking line search

```

1: INPUTS:
2:    $\mathbf{p}$  - search direction
3:    $\mathbf{g}$  - vector assembly of  $\delta_{\mathbf{u}} \mathcal{R}$ 
4:    $\mathbf{u}$  - state parameter vector
5: OUTPUT:
6:    $\alpha$  - Step length.
7:
8: function BLS( $\mathbf{p}, \mathbf{g}, \mathbf{u}$ )
9:    $\alpha := 1, \quad c_1 := 10^{-4}, \quad c_2 := 9/10$ 
10:    $\ell_0 := \text{assemble } \mathcal{R}(\mathbf{u})$ 
11:    $\ell_k := \text{assemble } \mathcal{R}(\mathbf{u} + \alpha \mathbf{p})$ 
12:   while  $\ell_k \geq \ell_0 + c_1 \alpha \mathbf{g}^\top \mathbf{p}$  do
13:      $\alpha := c_2 \alpha$ 
14:      $\ell_k := \text{assemble } \mathcal{R}(\mathbf{u} + \alpha \mathbf{p})$ 
15:   end while
16:   return  $\alpha$ 
17: end function

```

Chapter 7

Optimization with constraints

When the solution space for a problem is restricted by equality or inequality constraints, new theory is required to derive solutions. These problems can be stated in the form (Nocedal and Wright, 2000)

$$\min_{\vartheta \in \mathbb{R}^n} \mathcal{F}(\vartheta) \quad \text{subject to} \quad \begin{cases} \mathcal{R}(\vartheta) = 0, \\ c(x) \geq 0, \end{cases} \quad (7.1)$$

with parameter vector $\vartheta = [u \ x]^\top$. The real-valued functions $\mathcal{F}(\vartheta)$, $\mathcal{R}(\vartheta)$, and $c(x)$ are all smooth and defined on a subset of \mathbb{R}^n created from a finite-element discretization in one, two, or three dimensions. The function to be minimized, $\mathcal{F}(\vartheta)$, is referred to as the *objective* function with dependent *state* parameter $u \in \mathbb{R}^n$ and dependent *control* parameter $x \in \mathbb{R}^n$.

One method of solving problems of form (7.1) is through the use of a bounded version of Algorithm 2 referred to as `L_BFGS_B` (Byrd et al., 1995); however, a more modern and efficient class of constrained optimization algorithms known as *interior point* (IP) methods have been shown to perform quite well for problems of this type (Nocedal and Wright, 2000). Because CSLVR utilizes an IP method implemented by the FEniCS optimization software Dolfin-Adjoint (Farrell et al., 2013), this is the method described here.

For the applications presented in Part II of this manuscript, objective $\mathcal{F}(\vartheta)$ and constraint $\mathcal{R}(\vartheta)$ are *functionals*: the mapping from the space of functions to the space of real numbers, and so the following theory will be presented in this context. For examples of functionals, examine Chapters 5 and 6.

7.1 The control method

A stationary point for ϑ -optimization problem (7.1) is defined as one where an arbitrary change $\delta\mathcal{F}$ of objective \mathcal{F} caused by perturbations δu or δx in state and control parameter, respectively, lead to an increase in \mathcal{F} (Bryson and Ho, 1975). Thus it is necessary that

$$\frac{\delta\mathcal{F}}{\delta u} = 0, \quad \text{and} \quad \frac{\delta\mathcal{F}}{\delta x} = 0. \quad (7.2)$$

Using the chain rule of variations, the perturbations of \mathcal{F}

and \mathcal{R} in an arbitrary direction ϕ are

$$\frac{\delta\mathcal{F}}{\delta\phi} = \frac{\delta\mathcal{F}}{\delta u} \frac{\delta u}{\delta\phi} + \frac{\delta\mathcal{F}}{\delta x} \frac{\delta x}{\delta\phi}, \quad (7.3)$$

$$\frac{\delta\mathcal{R}}{\delta\phi} = \frac{\delta\mathcal{R}}{\delta u} \frac{\delta u}{\delta\phi} + \frac{\delta\mathcal{R}}{\delta x} \frac{\delta x}{\delta\phi}. \quad (7.4)$$

Because it is desired that $\delta_\phi\mathcal{R} = 0$, and with non-singular $\delta_u\mathcal{R}$, we can solve for $\delta_\phi u$ in (7.4),

$$\frac{\delta u}{\delta\phi} = -\frac{\delta u}{\delta\mathcal{R}} \frac{\delta\mathcal{R}}{\delta x} \frac{\delta x}{\delta\phi}. \quad (7.5)$$

We then insert (7.5) into (7.3),

$$\begin{aligned} \frac{\delta\mathcal{F}}{\delta\phi} &= -\frac{\delta\mathcal{F}}{\delta u} \left(\frac{\delta u}{\delta\mathcal{R}} \frac{\delta\mathcal{R}}{\delta x} \frac{\delta x}{\delta\phi} \right) + \frac{\delta\mathcal{F}}{\delta x} \frac{\delta x}{\delta\phi} \\ &= \left(\frac{\delta\mathcal{F}}{\delta x} - \frac{\delta\mathcal{F}}{\delta u} \frac{\delta u}{\delta\mathcal{R}} \frac{\delta\mathcal{R}}{\delta x} \right) \frac{\delta x}{\delta\phi}, \end{aligned} \quad (7.6)$$

and thus because we require $\delta_\phi\mathcal{F} = 0$ for any non-zero $\delta_\phi x$,

$$\begin{aligned} \frac{\delta\mathcal{F}}{\delta x} - \frac{\delta\mathcal{F}}{\delta u} \frac{\delta u}{\delta\mathcal{R}} \frac{\delta\mathcal{R}}{\delta x} &= 0 \\ \frac{\delta\mathcal{F}}{\delta x} + \lambda \frac{\delta\mathcal{R}}{\delta x} &= 0, \end{aligned} \quad (7.7)$$

where *Lagrange multiplier* or *adjoint variable* λ adjoins constraint functional \mathcal{R} to objective functional \mathcal{F} , and is given by

$$-\lambda = \frac{\delta\mathcal{F}}{\delta u} \frac{\delta u}{\delta\mathcal{R}} = \frac{\delta\mathcal{F}}{\delta\mathcal{R}} \Big|_u. \quad (7.8)$$

Therefore, λ is the direction of decent of objective \mathcal{F} with respect to constraint \mathcal{R} at a given energy state u .

It is now convenient to define the *Lagrangian*

$$\mathcal{L}(u, x, \lambda) = \mathcal{F}(u, x) + (\lambda, \mathcal{R}(u, x)), \quad (7.9)$$

where the notation $(f, g) = \int_\Omega f g d\Omega$ is the inner product. Using Lagrangian (7.9), the first necessary condition in (7.2) is satisfied when λ is chosen – say $\lambda = \lambda^*$ – such that for a given state u and control parameter x ,

$$\lambda^* = \operatorname{argmin}_\lambda \left\| \frac{\delta}{\delta u} \mathcal{L}(u, x; \lambda) \right\|. \quad (7.10)$$

This λ^* may then be used in condition (7.7) to calculate the direction of decent of Lagrangian (7.9) with respect to the control variable x for a given state u and adjoint variable λ^* ,

$$G = \frac{\delta}{\delta x} \mathcal{L}(u, x, \lambda^*). \quad (7.11)$$

This *Gâteaux derivative*, or first variation of Lagrangian \mathcal{L} with respect to x (see §6.1.2), provides a direction which control parameter x may follow in order to satisfy the second condition in (7.2) and thus minimize objective functional \mathcal{F} .

7.2 Log-barrier solution process

To determine a locally optimal value of u , a variation of a primal-dual-interior-point algorithm with a filter-line-search method may be used, as implemented by the IPOPT framework (Wächter and Biegler, 2006). Briefly, the algorithm implemented by IPOPT computes approximate solutions to a sequence of barrier problems

$$\min_{x \in \mathbb{R}^n} \left\{ \varphi_\mu(u, x) = \mathcal{F}(u, x) - \mu \sum_{i=1}^n \ln(c^i(x)) \right\} \quad (7.12)$$

for a decreasing sequence of barrier parameters μ converging to zero, and n is the number of degrees of freedom of the mesh. Neglecting equality constraints on the control variables, the first-order *necessary* conditions – known as the Karush-Kuhn-Tucker (KKT) conditions – for barrier problem (7.12) are

$$\begin{aligned} G(\vartheta, \lambda) - \lambda_b &= 0, \\ \Theta Z e - \mu e &= 0, \\ \{c(\vartheta), \lambda_b\} &\geq 0, \end{aligned} \quad (7.13)$$

where $G(\vartheta, \lambda) = \delta_x \mathcal{L}$, λ_b is the Lagrange multiplier for the bound constraint $c(\vartheta) \geq 0$ in (7.1), $\Theta = \text{diag}(c(\vartheta))$, $Z = \text{diag}(\lambda_b)$, and $e = \text{ones}(n)$. The so-called ‘optimality error’ for barrier problem (7.12) is

$$E_\mu(\vartheta, \lambda_b) = \max \left\{ \frac{\|G(\vartheta, \lambda) - \lambda_b\|_\infty}{s_d}, \frac{\|\Theta Z e - \mu e\|_\infty}{s_e} \right\},$$

with scaling parameters $s_d, s_e \geq 1$. This error defines the algorithm termination criteria with $\mu = 0$,

$$E_0(\vartheta^*, \lambda_b^*) \leq \epsilon_{\text{tol}}, \quad (7.14)$$

for approximate solution $(\vartheta^*, \lambda_b^*)$ and user-provided error tolerance ϵ_{tol} .

The solution to (7.13) for a given μ is attained by applying a damped version of Newton’s method, whereby the sequence of iterates $(\vartheta^k, \lambda_b^k)$ for iterate $k \leq k_{\text{max}}$ solves the system

$$\begin{bmatrix} W^k & -I \\ Z^k & \Theta^k \end{bmatrix} \begin{bmatrix} d_x^k \\ d_{\lambda_b}^k \end{bmatrix} = - \begin{bmatrix} G(\vartheta^k, \lambda) - \lambda_b^k \\ \Theta^k Z^k e - \mu e \end{bmatrix}, \quad (7.15)$$

with Hessian matrix

$$W^k = \frac{\delta^2}{\delta x^k \delta x^k} \mathcal{L}(u^k, x^k, \lambda) = \frac{\delta}{\delta x^k} G(\vartheta^k, \lambda). \quad (7.16)$$

Once search directions $(d_x^k, d_{\lambda_b}^k)$ have been found, the subsequent iterate is computed from

$$x^{k+1} = x^k + \ell^k d_x^k \quad (7.17)$$

$$\lambda_b^{k+1} = \lambda_b^k + \ell^k d_{\lambda_b}^k, \quad (7.18)$$

with step sizes ℓ determined by a backtracking-line-search procedure similar to Algorithm 3 to enforce an analogous set of Wolfe conditions as (6.14, 6.15), while also requiring that a sufficient decrease in φ_μ in (7.12) be attained.

Finally, because φ_μ is dependent on objective \mathcal{F} , objective $\mathcal{F}(u^{k+1}, x^{k+1})$ must be evaluated for a series of potential control parameter x^{k+1} values in (7.17). Hence multiple solutions of constraint relation $\mathcal{R}(u^{k+1}, x^{k+1}) = 0$ in (7.1) are required, one for each potential state parameter u^{k+1} for a given x^{k+1} . Finally, at the end of each iteration, adjoint variable λ is determined by solving (7.10) and used to compute the next iteration’s Gâteaux derivative $G(x^{k+1}, \lambda)$. For further details, examine Wächter and Biegler (2006).

Part II

Dynamics of ice-sheets and glaciers

Chapter 8

Fundamentals of flowing ice

Large bodies of ice behave as a highly viscous and thermally-dependent system. The primary variables associated with an ice-sheet or glacier defined over a domain Ω with boundary Γ (see Figure 8.1) are velocity \mathbf{u} with components u , v , and w in the x , y , and z directions; pressure p ; and internal energy θ . These variables are inextricably linked by the fundamental conservation equations

$$-\nabla \cdot \sigma = \rho \mathbf{g} \quad \text{in } \Omega \quad \leftarrow \text{momentum} \quad (8.1)$$

$$\nabla \cdot \mathbf{u} = 0 \quad \text{in } \Omega \quad \leftarrow \text{mass} \quad (8.2)$$

$$\rho \dot{\theta} = -\nabla \cdot \mathbf{q} + Q \quad \text{in } \Omega \quad \leftarrow \text{energy}. \quad (8.3)$$

These relations are in turn defined with gravitational acceleration vector $\mathbf{g} = [0 \ 0 \ -g]^\top$, ice density ρ , energy flux \mathbf{q} , strain-heat Q , and Cauchy-stress tensor

$$\sigma = \tau - pI, \quad \tau = 2\eta \dot{\epsilon} \quad (8.4)$$

further defined with rank-two identity tensor I , shear viscosity η , and strain-rate tensor

$$\begin{aligned} \dot{\epsilon} &= 1/2 [\nabla \mathbf{u} + (\nabla \mathbf{u})^\top] \\ &= \begin{bmatrix} \dot{\epsilon}_{xx} & \dot{\epsilon}_{xy} & \dot{\epsilon}_{xz} \\ \dot{\epsilon}_{yx} & \dot{\epsilon}_{yy} & \dot{\epsilon}_{yz} \\ \dot{\epsilon}_{zx} & \dot{\epsilon}_{zy} & \dot{\epsilon}_{zz} \end{bmatrix} \\ &= \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} & \frac{1}{2} \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \\ \frac{1}{2} \left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \right) & \frac{1}{2} \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) & \frac{\partial w}{\partial z} \end{bmatrix}. \end{aligned} \quad (8.5)$$

Shear viscosity η is derived from Nye's generalization of Glen's flow law (Glen, 1952; Nye, 1957)

$$\dot{\epsilon} = A(\theta) \tau_e^{n-1} \tau, \quad (8.6)$$

defined with Glen's flow parameter n , the deviatoric part of Cauchy-stress tensor (8.4) $\tau = 2\eta \dot{\epsilon}$, and Arrhenius-type energy-dependent flow-rate factor $A(\theta)$.

The second invariant of full-stress-tensor (8.4) – referred to as the *effective stress* – is given by

$$\begin{aligned} \tau_e^2 &= \frac{1}{2} \text{tr}(\tau^2) = \frac{1}{2} [\tau_{ij} \tau_{ij}] \\ &= \frac{1}{2} [\tau_{xx}^2 + \tau_{yy}^2 + \tau_{zz}^2 + 2\tau_{xy}^2 + 2\tau_{xz}^2 + 2\tau_{yz}^2]. \end{aligned} \quad (8.7)$$

Likewise, the second invariant of strain-rate tensor (8.5) – known as the *effective strain-rate* – is given by

$$\begin{aligned} \dot{\epsilon}_e^2 &= \frac{1}{2} \text{tr}(\dot{\epsilon}^2) = \frac{1}{2} [\dot{\epsilon}_{ij} \dot{\epsilon}_{ij}] \\ &= \frac{1}{2} [\dot{\epsilon}_{xx}^2 + \dot{\epsilon}_{yy}^2 + \dot{\epsilon}_{zz}^2 + 2\dot{\epsilon}_{xy}^2 + 2\dot{\epsilon}_{xz}^2 + 2\dot{\epsilon}_{yz}^2]. \end{aligned} \quad (8.8)$$

Due to the fact that the viscosity of ice η is a scalar field, the strain-rate and stress-deviator tensors in (8.6) may be set equal to their invariants. Their relationship with viscosity η is then evaluated,

$$\dot{\epsilon}_e = A \tau_e^{n-1} \tau_e = A \tau_e^n \implies \tau_e = A^{-\frac{1}{n}} \dot{\epsilon}_e^{\frac{1}{n}}. \quad (8.9)$$

Inserting (8.9) into (8.6) and solving for τ results in

$$\begin{aligned} \tau &= A^{-1} \tau_e^{1-n} \dot{\epsilon} = A^{-1} \left(A^{-\frac{1}{n}} \dot{\epsilon}_e^{\frac{1}{n}} \right)^{1-n} \dot{\epsilon} \\ &= A^{-1} A^{\frac{n-1}{n}} \dot{\epsilon}_e^{\frac{1-n}{n}} \dot{\epsilon} = A^{-\frac{1}{n}} \dot{\epsilon}_e^{\frac{1-n}{n}} \dot{\epsilon}. \end{aligned}$$

Next, using deviatoric-stress-tensor definition (8.4),

$$\eta = \frac{1}{2} \tau \dot{\epsilon}^{-1} = \frac{1}{2} \left(A^{-\frac{1}{n}} \dot{\epsilon}_e^{\frac{1-n}{n}} \dot{\epsilon} \right) \dot{\epsilon}^{-1} = \frac{1}{2} A^{-\frac{1}{n}} \dot{\epsilon}_e^{\frac{1-n}{n}},$$

When solving discrete systems, a strain-regularization term $\dot{\epsilon}_0 \ll 1$ may be introduced to eliminate singularities in areas of low strain-rate (Pattyn, 2003); the resulting thermally-dependent viscosity is given by

$$\eta(\theta, \mathbf{u}) = \frac{1}{2} A(\theta)^{-1/n} (\dot{\epsilon}_e(\mathbf{u}) + \dot{\epsilon}_0)^{\frac{1-n}{n}}. \quad (8.10)$$

Finally, the strain-heating term Q in (8.3) is defined as the third invariant (the trace) of the tensor product of strain-rate tensor (8.5) and the deviatoric component of Cauchy-stress tensor (8.4), $\tau = 2\eta \dot{\epsilon}$ (Greve and Blatter, 2009)

$$Q(\theta, \mathbf{u}) = \text{tr}(\dot{\epsilon} \cdot \tau) = 2\eta \text{tr}(\dot{\epsilon}^2) = 4\eta \dot{\epsilon}_e^2. \quad (8.11)$$

Equations (8.1 – 8.3) and corresponding boundary conditions are described in the following chapters. FEniCS source code will be provided whenever possible, and are available through the open-source software *Cryospheric Problem Solver* (CSLVR), an expansion of the FEniCS software *Variational Glacier Simulator* (VarGlaS) developed by Brinkerhoff and Johnson (2013).

8.1 List of symbols

θ	J kg^{-1}	internal energy (10.1)
θ_m	J kg^{-1}	pressure-melting energy (10.13)
θ_c	J kg^{-1}	maximum energy (10.72)
$\bar{\theta}$	J kg^{-1}	enthalpy (10.19)
T	K	temperature (10.16)
T_m	K	pressure-melting temp. (10.12)
T_S	K	2-meter depth surface temp. (10.27)
W	—	water content (10.15)
W_c	—	maximum water content (10.10)
W_S	—	surface water content (10.31)
\mathbf{q}	kg s^{-3}	energy flux (10.3, 10.4, 10.18)
\mathbf{q}_s	kg s^{-3}	sensible heat flux (10.4)
\mathbf{q}_l	kg s^{-3}	latent heat flux (10.4)
ρ	kg m^{-3}	density (10.7)
k	$\text{J s}^{-1}\text{m}^{-1}\text{K}^{-1}$	mixture thermal conductivity (10.5)
k_i	$\text{J s}^{-1}\text{m}^{-1}\text{K}^{-1}$	thermal conductivity of ice (10.8)
k_0	—	non-advective transport coef. (10.18)
c	$\text{J kg}^{-1}\text{K}^{-1}$	mixture heat capacity (10.6)
c_i	$\text{J kg}^{-1}\text{K}^{-1}$	heat capacity of ice (10.9)
κ	$\text{J s}^{-1}\text{m}^{-1}\text{K}^{-1}$	enthalpy-gradient cond'v'ty (10.18)
ν	$\text{J m}^{-1}\text{s}^{-1}$	non-advective water-flux coef. (10.4)
p	Pa	pressure (8.4)
\mathbf{f}	Pa m^{-1}	volumetric body forces (8.1)
\mathbf{g}	m s^{-2}	gravitational acceleration vector
\mathbf{u}	m s^{-1}	velocity vector
\mathbf{n}	—	outward-normal vector
Q	$\text{J m}^{-3}\text{s}^{-1}$	internal friction (8.11)
Ξ	m^2s^{-1}	mixture diffusivity (10.21)
σ	Pa	Cauchy-stress tensor (8.4)
σ_{BP}	Pa	first-order stress tensor (9.26)
σ_{PS}	Pa	plane-strain stress tensor (9.36)
σ_{RS}	Pa	reform.-Stokes stress tensor (9.54)
τ	Pa	deviatoric-stress tensor (8.4)
$\dot{\epsilon}$	s^{-1}	rate-of-strain tensor (8.5)
$\dot{\epsilon}_e$	s^{-1}	effective strain-rate (8.8)
$\dot{\epsilon}_{BP}$	s^{-1}	first-order eff. strain-rate (9.21)
$\dot{\epsilon}_{PS}$	s^{-1}	plane-strain eff. strain-rate (9.37)
$\dot{\epsilon}_{RS}$	s^{-1}	reform.-Stokes eff. strain-rate (9.52)
η	Pa s	shear viscosity (8.10)
η_{BP}	Pa s	first-order shear viscosity (9.22)
η_{PS}	Pa s	plane-strain shear viscosity (9.38)
η_{RS}	Pa s	reform.-Stokes shear viscosity (9.53)
f_w	Pa	hydrostatic pressure (9.5)
f_e	Pa	exterior pressure (9.31)
f_c	Pa	cryostatic pressure (9.31)
β	$\text{kg m}^{-2}\text{s}^{-1}$	basal-sliding coefficient (9.3)
A	$\text{Pa}^{-3}\text{s}^{-1}$	flow-rate factor with $n = 3$ (10.22)
q_{geo}	$\text{J s}^{-1}\text{m}^{-2}$	geothermal heat flux
q_{fric}	$\text{J s}^{-1}\text{m}^{-2}$	frictional heating (10.25)
g_N	$\text{J s}^{-1}\text{m}^{-2}$	basal energy source (10.26)
M_b	m s^{-1}	basal melting rate (10.36)
F_b	m s^{-1}	basal water discharge (10.37)
S	m	atmospheric surface height
B	m	basal surface height
H	m	ice thickness

h	m	element diameter
τ_{IE}	$\text{s m}^3 \text{kg}^{-3}$	energy intr'sic-time par. (10.58)
τ_{BV}	—	balance vel. intr'sic-time par. (13.26)
τ_{age}	s	age intrinsic-time parameter (15.4)
P_ϵ	—	element Péclet number (10.58)
ξ	—	energy intr'sic-time coef. (10.59, 10.60)
α	—	temperate zone coefficient (10.41)
\mathbf{r}	J s^{-1}	energy residual vector (10.63)
\mathcal{C}	J s^{-1}	energy advection matrix (10.64)
\mathcal{K}	J s^{-1}	conductive gradient matrix (10.65)
\mathcal{D}	J s^{-1}	energy diffusion matrix (10.66)
\mathcal{S}	J s^{-1}	energy stabilization matrix (10.67)
\mathbf{f}^{ext}	J s^{-1}	ext. basal energy flux vec. (10.68)
\mathbf{f}^{int}	J s^{-1}	internal strain heat vector (10.69)
\mathbf{f}^{stz}	J s^{-1}	stabilization vector (10.70)
Ω	m^3	domain volume
Γ	m^2	domain outer surface
Γ_A	m^2	atmospheric surface
Γ_S	m^2	complete upper surface
Γ_C	m^2	cold grounded basal surface
Γ_T	m^2	temperate grounded basal surface
Γ_G	m^2	complete grounded basal surface
Γ_W	m^2	surface in contact with ocean
Γ_E	m^2	non-grounded surface
Γ_D	m^2	interior lateral surface
\mathcal{A}	J s^{-1}	momentum variational princ. (9.8)
\mathcal{A}_{BP}	J s^{-1}	first-order momentum principle (9.33)
\mathcal{A}_{PS}	J s^{-1}	plane-strain momentum pr'c'p. (9.45)
\mathcal{A}_{RS}	J s^{-1}	ref.-Stokes momentum princ. (9.62)
Λ	Pa	impen'bil'ty Lagrange mult. (9.12)
Λ_{BP}	Pa	BP impen'bil'ty Lagrange mult. (9.35)
Λ_{PS}	Pa	PS impen'bil'ty Lagrange mult. (9.46)
V	Pa	viscous dissipation (9.9)
V_{BP}	Pa	first-order viscous dissipation (9.34)
V_{PS}	Pa	plane-strain viscous dissipation (9.47)
V_{RS}	Pa	reform.-Stokes viscous diss. (9.59)
\mathcal{R}	Pa s^{-1}	energy balance residual (10.48)
\mathcal{J}	m^6s^{-4}	energy objective functional (10.72)
\mathcal{L}	m^6s^{-4}	energy Lagrangian functional (10.74)
λ_b	m^5s^{-1}	F_b inequality const. Lagrange mult.
\mathcal{D}	J kg^{-1}	optimal water energy discrepancy
λ	$\text{m}^4\text{kg}^{-1}\text{s}^{-1}$	energy adjoint variable (10.75)
\mathcal{H}	J s^{-1}	momentum Lagrangian (12.6, 12.10)
λ	m s^{-1}	momentum adjoint variable (12.6)
\mathcal{J}	J s^{-1}	momentum objective functional (12.1)
γ_1	$\text{kg m}^{-2}\text{s}^{-1}$	L^2 cost coefficient (12.1)
γ_2	J s^{-1}	logarithmic cost coefficient (12.1)
γ_3	$\text{m}^6\text{kg}^{-1}\text{s}^{-1}$	Tikhonov regularization coef. (12.1)
γ_4	$\text{m}^6\text{kg}^{-1}\text{s}^{-1}$	TV regularization coeff. (12.1)
φ_μ	m^6s^{-4}	energy barrier problem (10.80)
φ_ω	J s^{-1}	momentum barrier problem (12.7)
\mathbf{d}	—	imposed dir. of balance velocity (13.18)
$\bar{\mathbf{u}}$	m s^{-1}	balance velocity (13.1)
\bar{u}	m s^{-1}	balance velocity magnitude (13.14)
$\hat{\mathbf{u}}$	—	balance velocity direction (13.14)
N	Pa m	membrane-stress tensor (14.11)
M	Pa	membrane-stress bal. tensor (14.13)

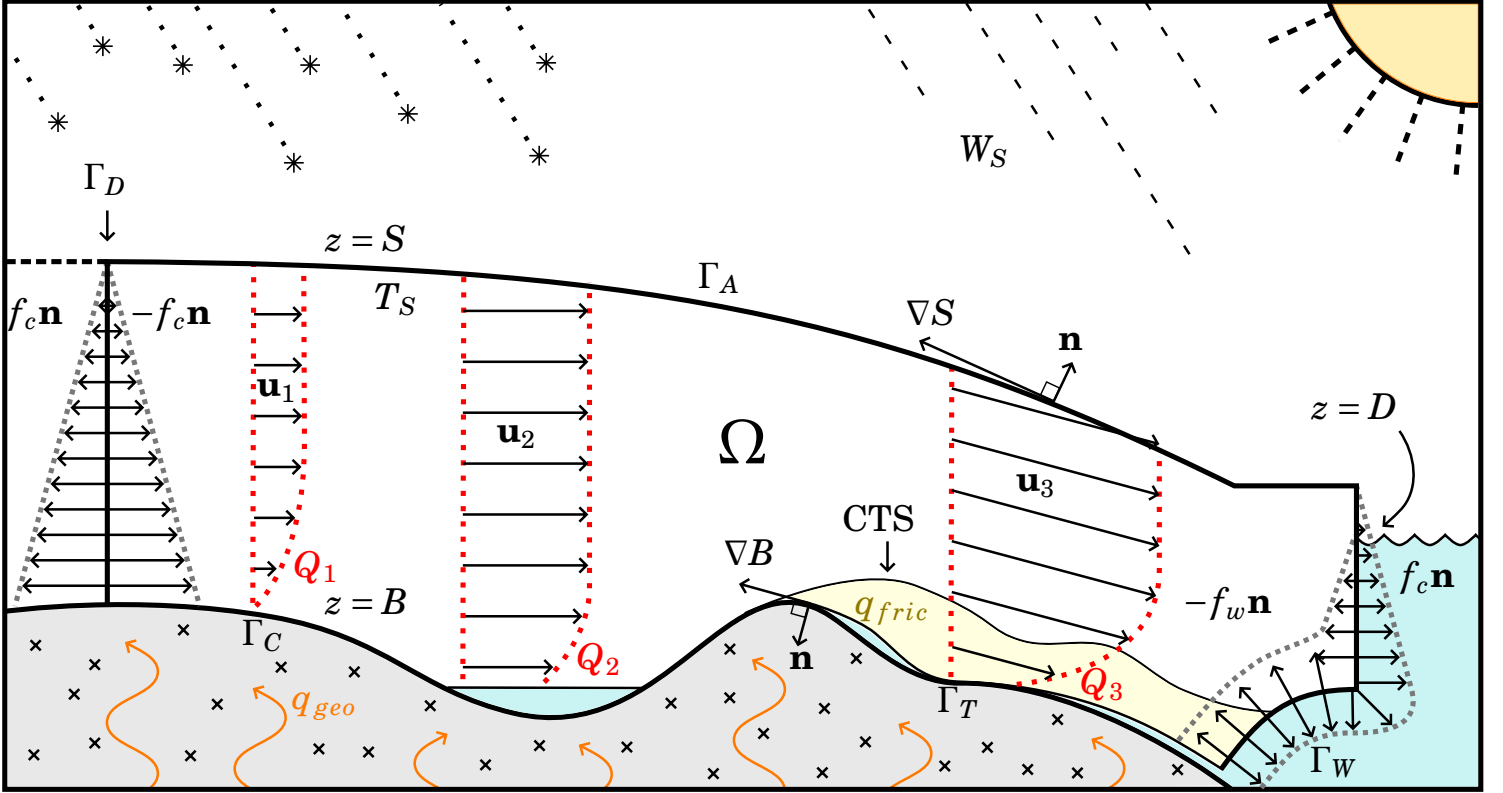


Figure 8.1: Illustration of thermo-dynamic processes for an ice-sheet with surface of height S on boundary Γ_A , grounded bed of height B with cold boundary Γ_C and temperate boundary Γ_T , ocean boundary Γ_W with ocean height D , interior ice lateral boundary Γ_D , interior volume Ω , outward-pointing normal vector \mathbf{n} , interior ice pressure normal to the boundary $f_c \mathbf{n}$, and water pressure exerted on the ice $-f_w \mathbf{n}$. Both the 2-meter depth average temperature T_S and water input W_S are used as surface boundary conditions, while the basal boundary is dependent upon energy-fluxes from geothermal sources q_{geo} and friction heat q_{fric} . The velocity profiles (dashed red) depend heavily on basal traction; for example, the basal traction associated with velocity profile \mathbf{u}_1 is very high. However, because the surface ice speed is small in regions far from the periphery of the ice-sheet, the gradient in velocity near the bed – and hence strain-heat Q_1 – is very low. Moreover, strain-heating may also be low if the basal traction is low, as is the case for strain heat Q_2 associated with velocity profile \mathbf{u}_2 flowing over a lake. Finally, observe that profile \mathbf{u}_3 flows over a temperate region formed from both increased friction and strain-heat Q_3 .

Table 8.1: Empirically-derived constants

g	9.81	m s^{-2}	gravitational acceleration
n	3	–	Glen's flow exponent
R	8.3144621	$\text{J mol}^{-1} \text{K}^{-1}$	universal gas constant
a	31556926	s a^{-1}	seconds per year
ρ_i	910	kg m^{-3}	density of ice
ρ_w	1000	kg m^{-3}	density of water
ρ_{sw}	1028	kg m^{-3}	density of seawater
k_w	0.561	$\text{J s}^{-1} \text{m}^{-1} \text{K}^{-1}$	thermal conductivity of water
c_w	4217.6	$\text{J kg}^{-1} \text{K}^{-1}$	heat capacity of water
L_f	3.34×10^5	J kg^{-1}	latent heat of fusion
T_w	273.15	K	triple point of water
γ	9.8×10^{-8}	K Pa^{-1}	pressure-melting coefficient

Chapter 9

Momentum and mass balance

Momentum-balance equation (8.1) and mass-balance equation (8.2), collectively referred to as the Stokes equations (see §3.2, §3.3, §4.2, §5.8, and the introductory chapter of Elman, Silvester, and Wathen (2005)), are completed with boundary conditions encompassing the entire outer surface $\Gamma = \Gamma_A \cup \Gamma_W \cup \Gamma_G$, with atmospheric boundary Γ_A , boundary in contact with ocean Γ_W , basal boundary in contact with bedrock Γ_G , and complete basal boundary including floating ice Γ_B (see Figure 8.1),

$$\boldsymbol{\sigma} \cdot \mathbf{n} = \mathbf{0} \quad \text{on } \Gamma_A \quad \leftarrow \text{stress-free surface} \quad (9.1)$$

$$\boldsymbol{\sigma} \cdot \mathbf{n} = -f_w \mathbf{n} \quad \text{on } \Gamma_W \quad \leftarrow \text{water pressure} \quad (9.2)$$

$$(\boldsymbol{\sigma} \cdot \mathbf{n})_{\parallel} = -\beta \mathbf{u} \quad \text{on } \Gamma_G \quad \leftarrow \text{basal traction} \quad (9.3)$$

$$\mathbf{u} \cdot \mathbf{n} = 0 \quad \text{on } \Gamma_B \quad \leftarrow \text{impenetrability}, \quad (9.4)$$

with outward-pointing normal vector to the boundary $\mathbf{n} = [n_x \ n_y \ n_z]^T$, and hydrostatic pressure

$$f_w = \rho_{sw} g(D - z), \quad z < D, \quad (9.5)$$

with seawater density ρ_{sw} and ocean height D . Tangential component of stress (9.3) – with tangential components denoted $(\mathbf{v})_{\parallel} = \mathbf{v} - (\mathbf{v} \cdot \mathbf{n})\mathbf{n}$ – is proportional to the basal velocity $\mathbf{u}|_{\Gamma_G}$ and basal-traction coefficient $\beta \geq 0$. Notice that traction boundary (9.3) and impenetrability boundary (9.4) are identical to slip-friction boundary conditions (3.17) and (3.16) explored previously in §3.3 and §5.8.

Throughout the following sections, Python source code associated with these fundamental equations will be provided. For example, viscosity (8.10) is created using FEniCS in Code Listing 9.1.

Code Listing 9.1: FEniCS code used to generate viscosity η as defined in the Momentum class, from which all of the momentum models of this chapter inherit.

```
def viscosity(self, U):
    """
    calculates the viscosity eta. Uses velocity vector <U> with
    components u,v,w. If <linear> == True, form viscosity from model.U3.
    """
    s = " ::: forming viscosity ::: "
    print_text(s, self.color())
    model = self.model
    n = model.n
    A_f = model.A_f
    eps_reg = model.eps_reg
    epsdot = self.effective_strain_rate(U)
    eta = 0.5 * A_f**(-1/n) * (epsdot + eps_reg)**((1-n)/(2*n))
    return eta
```

9.1 Full-Stokes equations

The expanded Stokes equations follow identically to the derivation of (4.12). These equations are

$$\frac{\partial}{\partial x} \left[2\eta \frac{\partial u}{\partial x} \right] - \frac{\partial p}{\partial x} + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] = 0 \quad (9.6a)$$

$$\frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \frac{\partial v}{\partial y} \right] - \frac{\partial p}{\partial y} + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right] = 0 \quad (9.6b)$$

$$\frac{\partial}{\partial x} \left[\eta \left(\frac{\partial w}{\partial x} + \frac{\partial u}{\partial z} \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial w}{\partial y} + \frac{\partial v}{\partial z} \right) \right] + \frac{\partial}{\partial z} \left[2\eta \frac{\partial w}{\partial z} \right] - \frac{\partial p}{\partial z} = \rho g, \quad (9.6c)$$

and conservation of mass relation (8.2),

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0. \quad (9.7)$$

Equations (9.6a, 9.6b, 9.6c, and 9.7) comprise a system of four equations and four unknowns u , v , w , and p . The complexity of solving this system and associated boundary conditions (9.1 – 9.4) has already been explored in §3.3 and §5.8; namely, the satisfaction or circumvention of inf-sup condition (3.15) and the correct imposition of Dirichlet condition (9.4). An elegant method satisfying these requirements is presented in the next section.

9.1.1 Variational principle

To solve system (8.1, 8.2, 9.1 – 9.4), the method described in Dukowicz, Price, and Lipscomb (2010) is used. This method makes use of a variational principle that uniquely determines velocity \mathbf{u} and pressure p by finding the extremum of the action

$$\begin{aligned} \mathcal{A}(\mathbf{u}, p) = & \int_{\Omega} (V(\dot{\epsilon}_e^2) - \rho \mathbf{g} \cdot \mathbf{u} - p \nabla \cdot \mathbf{u}) \, d\Omega \\ & + \int_{\Gamma_B} \left(\Lambda \mathbf{u} \cdot \mathbf{n} + \frac{1}{2} \beta \mathbf{u} \cdot \mathbf{u} \right) \, d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n} \cdot \mathbf{u} \, d\Gamma_L, \end{aligned} \quad (9.8)$$

with viscous-dissipation term

$$\begin{aligned}
 V(\dot{\epsilon}_e^2) &= \int_0^{\dot{\epsilon}_e^2} \eta(s) ds \\
 &= \frac{1}{2} A^{-1/n} \int_0^{\dot{\epsilon}_e^2} s^{\frac{1-n}{2n}} ds \\
 &= \frac{2n}{n+1} \left(\frac{1}{2} A^{-1/n} \right) (\dot{\epsilon}_e^2)^{\frac{n+1}{2n}} \\
 &= \frac{2n}{n+1} \left(\frac{1}{2} A^{-1/n} (\dot{\epsilon}_e^2)^{\frac{1-n}{2n}} \right) \dot{\epsilon}_e^2 \\
 &= \frac{2n}{n+1} \eta(\theta, \mathbf{u}) \dot{\epsilon}_e^2,
 \end{aligned} \tag{9.9}$$

where shear viscosity η is given by (8.10). Lagrange multiplier Λ enforces basal-surface impenetrability condition (9.4), while pressure p – defined as the mean compressive stress $p = -\sigma_{kk}/3$ – also takes on the role of a Lagrange multiplier to enforce incompressibility condition (8.2).

This extremum is defined as the solution to

$$\frac{\delta \mathcal{A}}{\delta \mathbf{u}} = 0, \quad \frac{\delta \mathcal{A}}{\delta p} = 0, \quad \frac{\delta \mathcal{A}}{\delta \Lambda} = 0, \tag{9.10}$$

and has been shown to be equivalent to the Stokes system (8.1, 8.2) by Dukowicz, Price, and Lipscomb (2010) and boundary conditions (9.1 – 9.4) by Dukowicz, Price, and Lipscomb (2011). It was later explained by Dukowicz, Price, and Lipscomb (2011) how the basal stress arising from Euler-Lagrange equations (9.10) is constrained to obey

$$\boldsymbol{\sigma} \cdot \mathbf{n}|_{\Gamma_B} = -\beta \mathbf{u} - \Lambda \mathbf{n}. \tag{9.11}$$

The magnitude of stress normal to the bed is determined by taking the dot product of (9.11) with \mathbf{n} , making use of bed-impenetrability condition (9.4), and the definition of a unit vector, resulting in

$$\Lambda = -\mathbf{n} \cdot \boldsymbol{\sigma} \cdot \mathbf{n}. \tag{9.12}$$

Therefore, Λ is equivalent to the magnitude of stress presented by the ice on the supporting bedrock. Relation (9.12) may be used to eliminate Lagrange multiplier Λ in (9.8); hence extremum conditions (9.10) are reduced to

$$\frac{\delta \mathcal{A}}{\delta \mathbf{u}} = 0, \quad \frac{\delta \mathcal{A}}{\delta p} = 0. \tag{9.13}$$

Additionally, by assuming that the magnitude of the normal component of deviatoric-stress tensor (8.4), $\mathbf{n} \cdot \boldsymbol{\tau} \cdot \mathbf{n}$, is much less than pressure p along the entire basal surface Γ_B , (9.12) simplifies to $\Lambda \approx p$. This approximation has in our experience lead to improved convergence characteristics of the discrete system when the topography includes steep basal gradients. Additionally, using both (9.11) and (9.12), observe that the tangential component of stress is

$$\begin{aligned}
 (\boldsymbol{\sigma} \cdot \mathbf{n})_{\parallel} &= \boldsymbol{\sigma} \cdot \mathbf{n} - (\mathbf{n} \cdot \boldsymbol{\sigma} \cdot \mathbf{n}) \mathbf{n} \\
 &= -\beta \mathbf{u} - \Lambda \mathbf{n} - (-\Lambda) \mathbf{n} \\
 &= -\beta \mathbf{u},
 \end{aligned} \tag{9.14}$$

and is thus consistent with traction-boundary-condition (9.3).

The source code of CSLVR uses an implementation similar to Code Listing 9.2.

Code Listing 9.2: CSLVR source code contained in the MomentumDukowiczStokes class.

```

# define variational problem :
U = Function(model.Q4, name = 'G')
dU = TrialFunction(model.Q4)
Phi = TestFunction(model.Q4)
du, dv, dw, dp = Phi
u, v, w, p = U

# create velocity vector :
U3 = as_vector([u,v,w])

# viscous dissipation :
epsdot = self.effective_strain_rate(U3)
if linear:
    s = " - using linear form of momentum using model.U3 in epsdot -"
    Uc = model.U3.copy(True)
    eta = self.viscosity(Uc)
    Vd = 2 * eta * epsdot
else:
    s = " - using nonlinear form of momentum -"
    eta = self.viscosity(U3)
    Vd = (2*n)/(n+1) * A_f*(-1/n) * (epsdot + eps_reg)**((n+1)/(2*n))
print_text(s, self.color())

# potential energy :
Pe = - rhoi * g * w

# dissipation by sliding :
Ut = U3 - dot(U3,N)*N
Sl = - 0.5 * beta * dot(Ut, Ut)

# incompressibility constraint :
Pc = p * div(U3)

# impenetrability constraint :
sig = self.stress_tensor(U3, p, eta)
lam = - dot(N, dot(sig, N))
Nc = - lam * (dot(U3, N) - Fb)

# pressure boundary :
Pb_w = - rhoiw*g*d + dot(U3, N)

# action :
A = + (Vd - Pe - Pc)*dx - Nc*dBed \
    - Sl*dBed_g - Pb_w*dBed_f - Pb_w*dLat_t

# the first variation of the action in the direction of a
# test function; the extremum :
self.mom_F = derivative(A, U, Phi)

# the first variation of the extremum in the direction
# a trial function; the Jacobian :
self.mom_Jac = derivative(self.mom_F, U, dU)

def stress_tensor(self, U, p, eta):
    """
    return the Cauchy stress tensor.
    """
    s = " ::: forming the Cauchy stress tensor ::: "
    print_text(s, self.color())

    I = Identity(3)
    tau = self.deviatoric_stress_tensor(U, eta)

    sigma = tau - p*I
    return sigma

def deviatoric_stress_tensor(self, U, eta):
    """
    return the deviatoric stress tensor.
    """
    s = " ::: forming the deviatoric part of the Cauchy stress tensor ::: "
    print_text(s, self.color())

    epi = self.strain_rate_tensor(U)
    tau = 2 * eta * epi
    return tau

def strain_rate_tensor(self, U):
    """
    return the strain-rate tensor of <U>.
    """
    epsdot = 0.5 * (grad(U) + grad(U).T)
    return epsdot

def effective_strain_rate(self, U):
    """
    return the effective strain rate squared.
    """
    epi = self.strain_rate_tensor(U)
    ep_xx = epi[0,0]
    ep_yy = epi[1,1]
    ep_zz = epi[2,2]
    ep_xy = epi[0,1]
    ep_xz = epi[0,2]
    ep_yz = epi[1,2]

    # Second invariant of the strain rate tensor squared
    epsdot = 0.5 * (ep_xx**2 + ep_yy**2 + ep_zz**2) \
        + ep_xy**2 + ep_xz**2 + ep_yz**2
    return epsdot

def default_solve_params(self):
    """
    Returns a set of default solver parameters that yield good performance
    """
    nparams = {'newton_solver' :
        {
            'linear_solver' : 'mumps',
            'relative_tolerance' : 1e-5,
            'relaxation_parameter' : 0.7,
            'maximum_iterations' : 25,
            'error_on_nonconvergence' : False,
        }
    }
    m_params = {'solver' : nparams}
    return m_params

def solve(self, annotate=False):
    """
    Perform the Newton solve of the full-Stokes equations
    """

```

```
# zero out self.velocity for good convergence for any subsequent solves,
# e.g. model.L_curve() :
model.assign_variable(self.get(U(), DOLFIN_EPS, cls=self)

# compute solution :
solve(self.mom_F == 0, self.U, J = self.mom_Jac, bcs = self.mom_bcs,
      annotate = annotate, solver_parameters = params['solver'])
u, v, w, p = self.U.split()
```

9.2 First-order approximation

Assumptions pertaining to both the state of stress and strain are appropriate over a large proportion of ice-sheets, and lead to considerable simplifications of full-Stokes equations (9.6). These simplifications and associated variational principle are described here.

9.2.1 Stress tensor simplification

The Stokes equations with four equations and four unknowns u , v , w , and p may be reduced to a system of three equations for the velocity components alone, as given by Blatter (1995) and Pattyn (2003). This is accomplished by first assuming that the shear stress components in the z -coordinate plane are negligible when compared to the z -coordinate normal stress, i.e. $\partial_x \sigma_{zx}, \partial_y \sigma_{zy} \ll \partial_z \sigma_{zz}$. Using this assumption, the final equation arising from the expansion of momentum-conservation relation (8.1), Equation (9.6c), is reduced to

$$\frac{\partial \sigma_{zz}}{\partial z} \approx \rho g, \quad (9.15)$$

which may be integrated from the surface to an arbitrary z -coordinate,

$$\int_z^S \frac{\partial \sigma_{zz}}{\partial z} dz' \approx \int_z^S \rho g dz'$$

$$\sigma_{zz}(S) - \sigma_{zz}(z) \approx \rho g(S - z).$$

Using surface-stress condition (9.1), $\sigma_{zz}(S) = 0$, and applying Cauchy-stress tensor definition (8.4),

$$p(z) \approx \rho g(S - z) + 2\eta \frac{\partial w}{\partial z}(z). \quad (9.16)$$

This pressure approximation may then be used to eliminate p from the remaining pressure derivative terms in momentum-balance (8.1) with

$$\frac{\partial p}{\partial i} \approx \rho g \frac{\partial S}{\partial i} + \frac{\partial}{\partial i} \left[2\eta \frac{\partial w}{\partial z} \right], \quad i \in x, y. \quad (9.17)$$

allowing the simplification of (9.6a) to

$$\begin{aligned} \frac{\partial}{\partial x} \left[2\eta \frac{\partial u}{\partial x} \right] - \frac{\partial p}{\partial x} + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] &= 0 \\ \frac{\partial}{\partial x} \left[2\eta \frac{\partial u}{\partial x} \right] - \rho g \frac{\partial S}{\partial x} + \frac{\partial}{\partial x} \left[2\eta \frac{\partial w}{\partial z} \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] & \\ + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] &= 0 \\ \frac{\partial}{\partial x} \left[2\eta \left(\frac{\partial u}{\partial x} - \frac{\partial w}{\partial z} \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] &= \rho g \frac{\partial S}{\partial x}, \end{aligned} \quad (9.18)$$

and (9.6b) to

$$\begin{aligned} \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \frac{\partial v}{\partial y} \right] - \frac{\partial p}{\partial y} + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right] &= 0 \\ \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \frac{\partial v}{\partial y} \right] - \rho g \frac{\partial S}{\partial y} + \frac{\partial}{\partial y} \left[2\eta \frac{\partial w}{\partial z} \right] & \\ + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right] &= 0 \\ \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \left(\frac{\partial v}{\partial y} - \frac{\partial w}{\partial z} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right] &= \rho g \frac{\partial S}{\partial y}, \end{aligned} \quad (9.19)$$

which combined with conservation of mass relation (9.7),

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \quad \text{in } \Omega$$

gives three equations and three unknowns u , v , and w .

9.2.2 Strain tensor simplification

Next, assuming the horizontal gradients of w are much less than the vertical gradient of the horizontal components of velocity, i.e. $\partial_x w \ll \partial_z u$ and $\partial_y w \ll \partial_z v$, and using (9.15), strain-rate tensor (8.5) is decoupled from vertical velocity w , resulting in the strain-rate quasi-tensor

$$\tilde{\epsilon} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \frac{\partial u}{\partial z} \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} & \frac{1}{2} \frac{\partial v}{\partial z} \end{bmatrix}. \quad (9.20)$$

Effective strain-rate (8.8) is also decoupled from w using the equivalent relation to conservation of mass relation (8.2), $\dot{\epsilon}_{zz} = -(\dot{\epsilon}_{xx} + \dot{\epsilon}_{yy})$,

$$\dot{\epsilon}_{\text{BP}}^2 = \tilde{\epsilon}_{xx}^2 + \tilde{\epsilon}_{yy}^2 + \tilde{\epsilon}_{xx}\tilde{\epsilon}_{yy} + \tilde{\epsilon}_{xx}^2 + \tilde{\epsilon}_{xx}^2 + \tilde{\epsilon}_{yy}^2. \quad (9.21)$$

Using first-order effective strain-rate (9.21), the associated first-order shear viscosity is

$$\eta_{\text{BP}}(\theta, \mathbf{u}_h) = \frac{1}{2} A(\theta)^{-1/n} (\dot{\epsilon}_{\text{BP}} + \dot{\epsilon}_0)^{\frac{1-n}{n}}, \quad (9.22)$$

where horizontal vector components are denoted $\mathbf{g}_h = [g_x \ g_y]^\top$.

Finally, inserting pressure derivative approximation (9.17) and first-order strain-rate quasi-tensor (9.20) into conservation of momentum relation (8.1), simplification (9.18) becomes

$$\begin{aligned} \frac{\partial}{\partial x} \left[2\eta \left(\frac{\partial u}{\partial x} - \frac{\partial w}{\partial z} \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial u}{\partial z} + \frac{\partial w}{\partial x} \right) \right] &= \rho g \frac{\partial S}{\partial x} \\ \frac{\partial}{\partial x} \left[2\eta \left(\frac{\partial u}{\partial x} + \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \frac{\partial u}{\partial z} \right] &= \rho g \frac{\partial S}{\partial x} \\ \frac{\partial}{\partial x} \left[2\eta \left(2 \frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[\eta \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \frac{\partial u}{\partial z} \right] &= \rho g \frac{\partial S}{\partial x} \end{aligned} \quad (9.23)$$

and simplification (9.19) becomes

$$\begin{aligned} \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \left(\frac{\partial v}{\partial y} - \frac{\partial w}{\partial z} \right) \right] + \frac{\partial}{\partial z} \left[\eta \left(\frac{\partial v}{\partial z} + \frac{\partial w}{\partial y} \right) \right] &= \rho g \frac{\partial S}{\partial y} \\ \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \left(\frac{\partial v}{\partial y} + \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \right) \right] + \frac{\partial}{\partial z} \left[\eta \frac{\partial v}{\partial z} \right] &= \rho g \frac{\partial S}{\partial y} \\ \frac{\partial}{\partial x} \left[\eta \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) \right] + \frac{\partial}{\partial y} \left[2\eta \left(2 \frac{\partial v}{\partial y} + \frac{\partial u}{\partial x} \right) \right] + \frac{\partial}{\partial z} \left[\eta \frac{\partial v}{\partial z} \right] &= \rho g \frac{\partial S}{\partial y}, \end{aligned} \quad (9.24)$$

two equations and two unknowns u and v . The first-order momentum balance is therefore

$$\nabla \cdot \sigma_{\text{BP}} = \rho g (\nabla S)_h \quad \text{in } \Omega, \quad (9.25)$$

with Blatter-Pattyn stress quasi-tensor

$$\sigma_{BP} = 2\eta_{BP} \begin{bmatrix} \left(2\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y}\right) & \frac{1}{2}\left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x}\right) & \frac{1}{2}\frac{\partial u}{\partial z} \\ \frac{1}{2}\left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y}\right) & \left(2\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x}\right) & \frac{1}{2}\frac{\partial v}{\partial z} \\ \frac{1}{2}\frac{\partial u}{\partial z} & \frac{1}{2}\frac{\partial v}{\partial z} & 0 \end{bmatrix}. \quad (9.26)$$

9.2.3 First-order vertical velocity and boundary conditions

Because vertical velocity w has been eliminated from conservation of momentum (8.1) through the creation of first-order momentum balance (9.25), this component of velocity may be computed directly by integrating conservation of mass (8.2) vertically, resulting in

$$w(z) = w(B) - \int_B^z \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) dz', \quad (9.27)$$

where the basal vertical velocity is determined directly from impenetrability condition (9.4),

$$w(B) = -\frac{u(B)n_x + v(B)n_y}{n_z}. \quad (9.28)$$

Finally, because both incompressibility (8.2) and impenetrability (9.4) are enforced by vertical velocity relation (9.27, 9.28), the remaining first-order boundary conditions are

$$\sigma_{BP} \cdot \mathbf{n} = f_e \mathbf{n}_h \quad \text{on } \Gamma_E \quad \leftarrow \text{exterior stress} \quad (9.29)$$

$$\sigma_{BP} \cdot \mathbf{n} = -\beta \mathbf{u}_h \quad \text{on } \Gamma_G \quad \leftarrow \text{basal traction}, \quad (9.30)$$

where exterior stress condition (9.29) is defined over exterior boundary $\Gamma_E = \Gamma_A \cup \Gamma_W$ with pressure f_e derived by combining pressure approximation (9.16) and first-order stress tensor (9.26) with water pressure (9.5),

$$f_e = f_c - f_w, \quad f_c = \rho g(S - z). \quad (9.31)$$

Note that boundaries located on the upper surface of the ice correspond with $f_e = 0$ and are thus stress-free, while cliff faces have $z \neq S$ and hence $f_c \neq 0$ (see Figure 8.1).

9.2.4 First-order variational principle

Also compiled by Dukowicz, Price, and Lipscomb (2011) is a first-order variational principle for first-order momentum balance (9.25) and associated boundary conditions (9.29, 9.30),

$$\frac{\delta \mathcal{A}_{BP}}{\delta \mathbf{u}_h} = 0, \quad \frac{\delta \mathcal{A}_{BP}}{\delta \Lambda_{BP}} = 0. \quad (9.32)$$

where

$$\begin{aligned} \mathcal{A}_{BP}(\mathbf{u}_h) = & \int_{\Omega} (V(\varepsilon_{BP}^2) + \rho g \mathbf{u}_h \cdot (\nabla S)_h) d\Omega \\ & + \int_{\Gamma_B} \left(\Lambda_{BP} \mathbf{u}_h \cdot \mathbf{n}_h + \frac{1}{2} \beta \mathbf{u}_h \cdot \mathbf{u}_h \right) d\Gamma_B \\ & + \int_{\Gamma_E} f_e \mathbf{n}_h \cdot \mathbf{u}_h d\Gamma_E, \end{aligned} \quad (9.33)$$

$$V(\varepsilon_{BP}^2) = \int_0^{\varepsilon_{BP}^2} \eta_{BP}(s) ds = \frac{2n}{n+1} \eta_{BP}(\theta, \mathbf{u}_h) \varepsilon_{BP}^2, \quad (9.34)$$

and Λ_{BP} defined similarly to (9.12),

$$\Lambda_{BP} = -\mathbf{n} \cdot \sigma_{BP} \cdot \mathbf{n} \approx 0. \quad (9.35)$$

The source code of CSLVR includes an implementation similar to Code Listing 9.3.

Code Listing 9.3: CSLVR source code contained in the MomentumDukowiczBP class.

```
# define variational problem :
U = Function(model.Q2, name='g')
dU = TrialFunction(model.Q2)
Phi = TestFunction(model.Q2)
du, psi = Phi
dv, dv = dU
u, v = U

# vertical velocity :
dw = TrialFunction(model.Q)
chi = TestFunction(model.Q)
w = Function(model.Q, name='u_f')

# viscous dissipation :
U3 = as_vector([u,v,0])
epsdot = self.effective_strain_rate(U3)
if linear:
    s = - using linear form of momentum using model.U3 in epsdot -"
    U3_c = model.U3.copy(True)
    eta = self.viscosity(U3_c)
    Vd = 2 * eta * epsdot
else:
    s = - using nonlinear form of momentum -"
    eta = self.viscosity(U3)
    Vd = (2*n)/(n+1) * A_f**(-1/n) * (epsdot + eps_reg)**((n+1)/(2*n))
print_text(s, self.color())

# potential energy :
Pe = - rhoi * g * (u*S.dx(0) + v*S.dx(1))

# dissipation by sliding :
Sl = - 0.5 * beta * (u**2 + v**2)

# pressure boundary :
Pb = (rhoi*g*(S - z) - rhosw*g*D) * (u*N[0] + v*N[1])

# action :
A = (Vd_gnd - Pe)*dx - Sl*dBed_g - Pb*dBed_f - Pb*dLat_t

# the first variation of the action in the direction of a
# test function; the extremum :
self.mom_F = derivative(A, U, Phi)

# the first variation of the extremum in the direction
# a trial function; the Jacobian :
self.mom_Jac = derivative(self.mom_F, U, dU)

self.w_F = (u.dx(0) + v.dx(1) + dw.dx(2))*chi*dx \
+ (u*N[0] + v*N[1] + dw*N[2] - Fb)*chi*dBed

def strain_rate_tensor(self, U):
    """
    return the Dukowicz 'Blatter-Pattyn' simplified strain-rate tensor of <U>.
    """
    u,v,w = U
    epi = 0.5 * (grad(U) + grad(U).T)
    epi02 = 0.5*u.dx(2)
    epi12 = 0.5*v.dx(2)
    epi22 = -u.dx(0) - v.dx(1) # incompressibility
    epsdot = as_matrix([[epi[0,0], epi[0,1], epi02],
                        [epi[1,0], epi[1,1], epi12],
                        [epi02, epi12, epi22]])
    return epsdot

def effective_strain_rate(self, U):
    """
    return the Dukowicz BP effective strain rate squared.
    """
    epi = self.strain_rate_tensor(U)
    ep_xx = epi[0,0]
    ep_yy = epi[1,1]
    ep_zz = epi[2,2]
    ep_xy = epi[0,1]
    ep_xz = epi[0,2]
    ep_yz = epi[1,2]

    # Second invariant of the strain rate tensor squared
    epsdot = ep_xx**2 + ep_yy**2 + ep_zz**2 + ep_xy**2 \
+ ep_xz**2 + ep_yz**2
    return epsdot

def default_solve_params(self):
    """
    Returns a set of default solver parameters that yield good performance
    """
    nparams = {'newton_solver' :
        {
            'linear_solver' : 'cg',
            'preconditioner' : 'hybre_amg',
            'relative_tolerance' : 1e-5,
            'relaxation_parameter' : 0.7,
            'maximum_iterations' : 25,
            'error_on_nonconvergence' : False,
            'krylov_solver' :
            {
                'monitor_convergence' : False,
            }
        }
    }
    return m_params

def solve_pressure(self, annotate=False):
    """
    Solve for the Dukowicz BP pressure to model.p.
    """
    p = project(rhoi*g*(S - z) + 2*eta*w.dx(2),
        annotate=annotate)

def solve_vert_velocity(self, annotate=False):
    """
    Perform the Newton solve of the first-order equations
    """
    aw = assemble(lhs(self.w_F))
    lw = assemble(rhs(self.w_F))
```

```

w_solver = LUSolver(self.solve_params['vert_solve_method'])
w_solver.solve(aw, self.w.vector(), Lw, annotate=annotate)

def solve(self, annotate=False):
    """
    Perform the Newton solve of the first-order equations
    """
    # zero out self.velocity for good convergence for any subsequent solves,
    # e.g. model.L_curve() :
    model.assign_variable(self.get_U(), DOLFIN_EPS, cls=self)

    # compute solution :
    solve(self.mom_F == 0, self.U, J = self.mom_Jac,
          annotate = annotate, solver_parameters = params['solver'])
    u, v = self.U.split()

```

9.3 Plane-strain approximation

Many observations of the ice lie along x, y -coordinate transects. In order to explain these observations, the *plane-strain* momentum balance model (Hill, 1950) has been formulated for ice, and is based on the assumption that longitudinal stress and lateral shear are present only in the direction of velocity \mathbf{u} . Using this model with flow specified in the x -direction, all y -component terms of stress tensor (8.4) and strain-rate tensor (8.5) are eliminated, producing the two-dimensional model tensors

$$\sigma_{\text{PS}} = \begin{bmatrix} \sigma_{xx} & \sigma_{xz} \\ \sigma_{zx} & \sigma_{zz} \end{bmatrix}, \quad \text{and} \quad \tilde{\epsilon} = \begin{bmatrix} \tilde{\epsilon}_{xx} & \tilde{\epsilon}_{xz} \\ \tilde{\epsilon}_{zx} & \tilde{\epsilon}_{zz} \end{bmatrix}. \quad (9.36)$$

Effective strain-rate (8.8) is therefore reduced to

$$\dot{\epsilon}_{\text{PS}}^2 = \frac{1}{2} \left[\tilde{\epsilon}_{xx}^2 + \tilde{\epsilon}_{zz}^2 + 2\tilde{\epsilon}_{xz}^2 \right], \quad (9.37)$$

which is used within the plane-strain viscosity

$$\eta_{\text{PS}}(\theta, \mathbf{u}_p) = \frac{1}{2} A(\theta)^{-1/n} (\dot{\epsilon}_{\text{PS}} + \dot{\epsilon}_0)^{\frac{1-n}{n}}, \quad (9.38)$$

with xz -plane velocity $\mathbf{u}_p = [u \ w]^T$. The plane-strain Stokes system analogous to full-Stokes system (8.1, 8.2, 9.1 – 9.4) with stress tensor $\sigma_{\text{PS}} = 2\eta_{\text{PS}}\tilde{\epsilon} - p\mathbf{I}$ consists of

$$-\nabla \cdot \sigma_{\text{PS}} = \rho \mathbf{g}_p \quad \text{in } \Omega \quad \leftarrow \text{momentum} \quad (9.39)$$

$$\nabla \cdot \mathbf{u}_p = 0 \quad \text{in } \Omega \quad \leftarrow \text{mass} \quad (9.40)$$

$$\sigma_{\text{PS}} \cdot \mathbf{n}_p = \mathbf{0}_p \quad \text{on } \Gamma_A \quad \leftarrow \text{stress-free surface} \quad (9.41)$$

$$\sigma_{\text{PS}} \cdot \mathbf{n}_p = -f_w \mathbf{n}_p \quad \text{on } \Gamma_W \quad \leftarrow \text{water pressure} \quad (9.42)$$

$$(\sigma_{\text{PS}} \cdot \mathbf{n}_p)_{\parallel} = -\beta \mathbf{u}_p \quad \text{on } \Gamma_G \quad \leftarrow \text{basal traction} \quad (9.43)$$

$$\mathbf{u}_p \cdot \mathbf{n}_p = 0 \quad \text{on } \Gamma_B \quad \leftarrow \text{impenetrability}, \quad (9.44)$$

with outward-pointing normal vector to the boundary $\mathbf{n}_p = [n_x \ n_z]^T$, gravitational acceleration vector $\mathbf{g}_p = [0 \ -g]^T$, water pressure f_w as defined by (9.5), and basal-traction coefficient $\beta \geq 0$.

9.3.1 Plane-strain variational principle

Proceeding in an identical fashion as §9.1.1 and §9.2.4, the associated action for plane-strain momentum balance (9.39 –

9.44) is

$$\begin{aligned} \mathcal{A}_{\text{PS}}(\mathbf{u}_p, p) = & \int_{\Omega} (V(\dot{\epsilon}_{\text{PS}}^2) - \rho \mathbf{g}_p \cdot \mathbf{u}_p - p \nabla \cdot \mathbf{u}_p) d\Omega \\ & + \int_{\Gamma_B} \left(\Lambda_{\text{PS}} \mathbf{u}_p \cdot \mathbf{n}_p + \frac{1}{2} \beta \mathbf{u}_p \cdot \mathbf{u}_p \right) d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n}_p \cdot \mathbf{u}_p d\Gamma_L, \end{aligned} \quad (9.45)$$

where Λ_{PS} is defined similarly to (9.12) and (9.35),

$$\Lambda_{\text{PS}} = -\mathbf{n}_p \cdot \sigma_{\text{PS}} \cdot \mathbf{n}_p \approx p. \quad (9.46)$$

and with viscous dissipation term $V(\dot{\epsilon}_{\text{PS}}^2)$ defined from the same process leading to (9.9) and (9.34),

$$V(\dot{\epsilon}_{\text{PS}}^2) = \int_0^{\dot{\epsilon}_{\text{PS}}^2} \eta_{\text{PS}}(s) ds = \frac{2n}{n+1} \eta_{\text{PS}}(\theta, \mathbf{u}_p) \dot{\epsilon}_{\text{PS}}^2. \quad (9.47)$$

Finally, the extremum of action (9.45) is given by the solution (\mathbf{u}_p, p) of

$$\frac{\delta \mathcal{A}_{\text{PS}}}{\delta \mathbf{u}_p} = 0, \quad \frac{\delta \mathcal{A}_{\text{PS}}}{\delta p} = 0, \quad (9.48)$$

and are equivalent to Euler-Lagrange plane-strain Stokes equations and boundary conditions (9.39 – 9.44).

The source code of CSLVR uses an implementation similar to Code Listing 9.4.

Code Listing 9.4: CSLVR source code contained in the `MomentumDukowiczPlaneStrain` class.

```

# define variational problem :
U = Function(model.Q3, name = 'G')
dU = TrialFunction(model.Q3)
Phi = TestFunction(model.Q3)
phi, xsi, kappa = Phi
du, dw, dP = dU
u, w, p = U

# create velocity vector :
U2 = as_vector([u, w])

# viscous dissipation :
epsdot = self.effective_strain_rate(U2)
if linear:
    s = " - using linear form of momentum using model.U3 in epsdot -"
    U3_c = model.U3.copy(True)
    U3_2 = as_vector([U3_c[0], U3_c[1]])
    eta = self.viscosity(U3_2)
    Vd = 2 * eta * epsdot
else:
    s = " - using nonlinear form of momentum -"
    eta = self.viscosity(U2)
    Vd = (2*n)/(n+1) * A_f**(-1/n) * (epsdot + eps_reg)**((n+1)/(2*n))
print_text(s, self.color())

# potential energy :
Pe = - rhoi * g * w

# dissipation by sliding :
Ut = U2 - dot(U2, N)*N
Sl = - 0.5 * beta * dot(Ut, Ut)

# incompressibility constraint :
Pc = p * div(U2)

# impenetrability constraint :
sig = self.stress_tensor(U2, p, eta)
lam = - dot(N, dot(sig, N))
Nc = - lam * (dot(U2, N) - Fb)

# pressure boundary :
Pb_v = - rhoiw*g*D + dot(U2, N)
Pb_l = - rhoi*g*(S - z) * dot(U2, N)

# action :
A = (Vd - Pe - Pc)*dx - Nc*dBed \
    - Sl*dBed_g - Pb_w*dBed_f - Pb_w*dLat_t

# add lateral boundary conditions :
if use_lat_bcs:
    s = " - using internal divide lateral stress natural boundary " + \
        " conditions -"
    print_text(s, self.color())
    U3_c = model.U3.copy(True)
    U3_2 = as_vector([U3_c[0], U3_c[1]])
    eta_l = self.viscosity(U3_2)
    sig_l = self.stress_tensor(U3_2, model.p, eta_l)
    sig_l_1 = self.stress_tensor(U2, p, eta)
    A -= dot(dot(sig_l_1, N), U2) * dLat_d

# the first variation of the action in the direction of a
# test function; the extremum :
self.mom_F = derivative(A, U, Phi)

```



```

# the first variation of the extremum in the direction
# a trial function; the Jacobian :
self.mom_Jac = derivative(self.mom_F, U, du)

def strain_rate_tensor(self, U):
    """
    return the strain-rate tensor of self.U.
    """
    epsdot = 0.5 * (grad(U) + grad(U).T)
    return epsdot

def effective_strain_rate(self, U):
    """
    return the effective strain rate squared.
    """
    epi = self.strain_rate_tensor(U)
    ep_xx = epi[0,0]
    ep_zz = epi[1,1]
    ep_xz = epi[0,1]

    # Second invariant of the strain rate tensor squared
    epsdot = 0.5 * (ep_xx**2 + ep_zz**2) + ep_xz**2
    return epsdot

def default_solve_params(self):
    """
    Returns a set of default solver parameters that yield good performance
    """
    nparams = {'newton_solver' : {'linear_solver' : 'mumps',
                                'relative_tolerance' : 1e-5,
                                'relaxation_parameter' : 0.7,
                                'maximum_iterations' : 25,
                                'error_on_nonconvergence' : False}}

    m_params = {'solver' : nparams}
    return m_params

def solve(self, annotate=False):
    """
    Perform the Newton solve of the full-Stokes equations
    """
    # zero out self.velocity for good convergence for any subsequent solves,
    # e.g. model.L_curve() :
    model.assign_variable(self.get_U(), DOLFIN_EPS, cls=self)

    # compute solution :
    solve(self.mom_F == 0, self.U, J = self.mom_Jac, bcs = self.mom_bcs,
          annotate = annotate, solver_parameters = params['solver'])
    u, w, p = self.U.split()

```

9.4 Reformulated full-Stokes

A novel method introduced by Dukowicz (2012) utilized the foundation built by the action principles presented in Dukowicz, Price, and Lipscomb (2010) and Dukowicz, Price, and Lipscomb (2011). This method specifies the use of a velocity trial function that satisfies continuity equation (8.2) and impenetrability condition (9.4), and results in the elimination of Lagrange multipliers p and Λ in action (9.8). A version of this method has been incorporated into the CSLVR code, and varies only slightly from that presented by Dukowicz (2012).

The first step in generating the velocity trial space is to express vertical velocity component w in terms of the horizontal velocity components u and v , in a fashion similar to (9.27). To this end, we solve the first-order BVP for the vertical velocity component w^h

$$\nabla \cdot \mathbf{u}_w = 0 \quad \text{in } \Omega \quad (9.49a)$$

$$\mathbf{u}_w \cdot \mathbf{n} = 0 \quad \text{on } \Gamma_B, \quad (9.49b)$$

where $\mathbf{u}_w = [u \ v \ w^h]^T$ is reformulated velocity vector with previously computed horizontal velocity components u and v . The associated variational problem for (9.49) reads: find $w^h \in S_E^h \subset \mathcal{H}^1(\Omega)$ (see trial space (1.10)) such that

$$\int_{\Omega} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w^h}{\partial z} \right) \chi \, d\Omega + \int_{\Gamma_B} (u n_x + v n_y + w^h n_z) \chi \, d\Gamma_B = 0, \quad (9.50)$$

for all $\chi \in S_0^h \subset \mathcal{H}^1(\Omega)$ (see test space (1.11)). This system must be numerically calculated in tandem with the process determining the horizontal velocity components via a fixed-point or Picard iteration.

Next, strain-rate tensor (8.5) is expressed in terms of reformulated velocity \mathbf{u}_w ,

$$\tilde{\epsilon} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \left(\frac{\partial u}{\partial z} + \frac{\partial w^h}{\partial x} \right) \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} & \frac{1}{2} \left(\frac{\partial v}{\partial z} + \frac{\partial w^h}{\partial y} \right) \\ \frac{1}{2} \left(\frac{\partial w^h}{\partial x} + \frac{\partial u}{\partial z} \right) & \frac{1}{2} \left(\frac{\partial w^h}{\partial y} + \frac{\partial v}{\partial z} \right) & - \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \end{bmatrix}, \quad (9.51)$$

where incompressibility constraint (8.2) has been used to express the zz -component. The second invariant of this tensor provides the reformulated-Stokes effective strain-rate

$$\begin{aligned} \dot{\epsilon}_{RS}^2 &= \frac{1}{2} \text{tr}(\tilde{\epsilon}^2) = \frac{1}{2} \left[\tilde{\epsilon}_{ij} \tilde{\epsilon}_{ij} \right] \\ &= \frac{1}{2} \left[\tilde{\epsilon}_{xx}^2 + \tilde{\epsilon}_{yy}^2 + \tilde{\epsilon}_{zz}^2 + 2\tilde{\epsilon}_{xy}^2 + 2\tilde{\epsilon}_{xz}^2 + 2\tilde{\epsilon}_{yz}^2 \right], \end{aligned} \quad (9.52)$$

and reformulated-Stokes shear viscosity derived identically to viscosity (8.10),

$$\eta_{RS}(\theta, \mathbf{u}_w) = \frac{1}{2} A(\theta)^{-1/n} (\dot{\epsilon}_{RS} + \dot{\epsilon}_0)^{\frac{1-n}{n}}. \quad (9.53)$$

To eliminate the pressure dependence on the momentum balance we assume that the pressure is entirely cryostatic, such that $p = f_c = \rho g(S - z)$. It follows from the same procedure used to derive first-order quasi-stress tensor (9.26) that the reformulated-Stokes stress tensor under these assumptions is

$$\sigma_{RS} = 2\eta_{RS} \begin{bmatrix} \left(2\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \left(\frac{\partial u}{\partial z} + \frac{\partial w^h}{\partial x} \right) \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \left(2\frac{\partial v}{\partial y} + \frac{\partial u}{\partial x} \right) & \frac{1}{2} \left(\frac{\partial v}{\partial z} + \frac{\partial w^h}{\partial y} \right) \\ \frac{1}{2} \left(\frac{\partial w^h}{\partial x} + \frac{\partial u}{\partial z} \right) & \frac{1}{2} \left(\frac{\partial w^h}{\partial y} + \frac{\partial v}{\partial z} \right) & - \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) - f_c \end{bmatrix}. \quad (9.54)$$

Using this stress-tensor definition in place of σ in momentum-balance (8.1), while making use of the facts that $\partial_z f_c = -\rho g$ and $\partial_z S = 0$, result in the reformulated momentum balance $\nabla \cdot \sigma_{RS} = \rho g \nabla S$. Therefore, the complete reformulated-Stokes momentum balance analogous to full-Stokes system (8.1, 8.2, 9.1 – 9.4) and first-order system (9.25, 9.29, 9.30) consists of

$$\nabla \cdot \sigma_{RS} = \rho g \nabla S \quad \text{in } \Omega \quad \leftarrow \text{momentum} \quad (9.55)$$

$$\sigma_{RS} \cdot \mathbf{n} = (f_c - f_w) \mathbf{n} \quad \text{on } \Gamma_E \quad \leftarrow \text{exterior pressure} \quad (9.56)$$

$$(\sigma_{RS} \cdot \mathbf{n})_{\parallel} = -\beta \mathbf{u}_w \quad \text{on } \Gamma_G \quad \leftarrow \text{basal traction} \quad (9.57)$$

with exterior boundary $\Gamma_E = \Gamma_A \cup \Gamma_W$, outward-pointing normal vector to the boundary $\mathbf{n} = [n_x \ n_y \ n_z]^T$, gravitational acceleration vector $\mathbf{g}_p = [0 \ 0 \ -g]^T$, water pressure f_w as defined by (9.5), cryostatic pressure $f_c(z) = p(z) = \rho g(S - z)$, and basal-traction coefficient $\beta \geq 0$.

Note once again that the solution of reformulated system (9.49, 9.55 – 9.57) requires a fixed-point iteration whereby at iterate k , vertical velocity w_k^h is coupled to a given horizontal velocity solution u_{k-1}, v_{k-1} via variational problem (9.50). See §9.6 for details of the implementation used by CSLVR to accomplish this coupling.

9.4.1 Reformulated-Stokes variational principle

Proceeding in an identical fashion as §9.1.1, §9.2.4, and §9.3.1, the associated action for reformulated-Stokes system (9.39 – 9.44) is

$$\begin{aligned} \mathcal{A}_{\text{RS}}(\mathbf{u}_w, p) = & \int_{\Omega} (V(\dot{\epsilon}_{\text{RS}}^2) - \rho \mathbf{g} \cdot \mathbf{u}_w) d\Omega \\ & + \int_{\Gamma_B} \frac{1}{2} \beta \mathbf{u}_w \cdot \mathbf{u}_w d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n} \cdot \mathbf{u}_w d\Gamma_L, \end{aligned} \quad (9.58)$$

with viscous dissipation term $V(\dot{\epsilon}_{\text{RS}}^2)$ defined from the same process leading to (9.9), (9.34), and (9.47),

$$V(\dot{\epsilon}_{\text{RS}}^2) = \int_0^{\dot{\epsilon}_{\text{RS}}^2} \eta_{\text{RS}}(s) ds = \frac{2n}{n+1} \eta_{\text{RS}}(\theta, \mathbf{u}_w) \dot{\epsilon}_{\text{RS}}^2. \quad (9.59)$$

Reformulated-Stokes action (9.58) was simplified in Appendix A of Dukowicz (2012) by forming the expression for the gravitational work term

$$\int_{\Omega} \rho \mathbf{g} \cdot \mathbf{u}_w d\Omega = \int_{\Omega} \rho g \mathbf{u}_h \cdot (\nabla S)_h d\Omega, \quad (9.60)$$

where horizontal vector components are denoted $\mathbf{g}_h = [g_x \ g_y]^\top$. Additionally, a basal vertical velocity term analogous to expression (9.28) derived from impenetrability condition (9.49b) is used to reduce the basal-traction term in (9.58) to the expression

$$\int_{\Gamma_B} \frac{1}{2} \beta \mathbf{u}_w \cdot \mathbf{u}_w d\Gamma_B = \int_{\Gamma_B} \frac{1}{2} \beta \left(u^2 + v^2 + \left(\frac{un_x + vn_y}{n_z} \right)^2 \right) d\Gamma_B. \quad (9.61)$$

Inserting (9.60) and (9.61) into (9.58) results in the final reformulated-Stokes action

$$\begin{aligned} \mathcal{A}_{\text{RS}}(\mathbf{u}_w, p) = & \int_{\Omega} (V(\dot{\epsilon}_{\text{RS}}^2) - \rho g \mathbf{u}_h \cdot (\nabla S)_h) d\Omega \\ & + \int_{\Gamma_B} \frac{1}{2} \beta \left(\mathbf{u}_h \cdot \mathbf{u}_h + \left(\frac{\mathbf{u}_h \cdot \mathbf{n}_h}{n_z} \right)^2 \right) d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n} \cdot \mathbf{u}_w d\Gamma_L, \end{aligned} \quad (9.62)$$

with extremum given by

$$\frac{\delta \mathcal{A}_{\text{RS}}}{\delta \mathbf{u}_h} = 0, \quad (9.63)$$

and produces the unique minimizer (\mathbf{u}_h, p) . It was also shown by Dukowicz (2012) that (9.63) is equivalent to reformulated-Stokes Euler-Lagrange momentum equations and boundary conditions (9.39 – 9.43).

The source code of CSLVR uses an implementation similar to Code Listing 9.5; note the use of Newton-Raphson Code Listing 6.1 in the solve method.

Code Listing 9.5: CSLVR source code contained in the MomentumDukowiczStokesReduced class.

```
# define variational problem :
U = Function(model.Q2, name = 'Q')
dU = TrialFunction(model.Q2)
Phi = TestFunction(model.Q2)
phi, psi = Phi
du, dv = dU
u, v = U

# vertical velocity :
dw = TrialFunction(model.Q)
chi = TestFunction(model.Q)
w = Function(model.Q, name='w_f')
self.w_F = (u.dx(0) + v.dx(1) + dw.dx(2)) * chi * dx \
+ (u * N[0] + v * N[1] + dw * N[2] - Fb) * chi * dBed

# viscous dissipation :
U3 = as_vector([u, v, model.w])
epsdot = self.effective_strain_rate(U3)
if linear:
    s = " - using linear form of momentum using model.U3 in epsdot -"
    Uc = model.U3.copy(True)
    eta = self.viscosity(Uc)
    Vd = 2 * eta * epsdot
else:
    s = " - using nonlinear form of momentum -"
    eta = self.viscosity(U3)
    Vd = (2 * n) / (n + 1) * A.f * (-1 / n) * (epsdot + eps_reg) * ((n + 1) / (2 * n))
print_text(s, self.color())

# potential energy :
Pe = - rho * g * (u * S.dx(0) + v * S.dx(1))

# dissipation by sliding :
w_b = (Fb - u * N[0] - v * N[1]) / N[2]
Sl = - 0.5 * beta * (u * 2 + v * 2 + w_b * 2)

# pressure boundary :
Pb = (rho * g * (S - z) - rho * w * g * D) * dot(U3, N)

# action :
A = + Vd * dx - Pe * dx - Sl * dBed_g - Pb * dBed_f - Pb * v * dLat_t

# the first variation of the action in the direction of a
# test function; the extremum :
self.mom_F = derivative(A, U, Phi)

# the first variation of the extremum in the direction
# a trial function; the Jacobian :
self.mom_Jac = derivative(self.mom_F, U, dU)

def strain_rate_tensor(self, U):
    """
    return the strain-rate tensor for the velocity <U>.
    """
    u, v, w = U
    epi = 0.5 * (grad(U) + grad(U).T)
    epi22 = -u.dx(0) - v.dx(1) # incompressibility
    epsdot = as_matrix([[epi[0,0], epi[0,1], epi[0,2]],
                        [epi[1,0], epi[1,1], epi[1,2]],
                        [epi[2,0], epi[2,1], epi22]])
    return epsdot

def effective_strain_rate(self, U):
    """
    return the effective strain rate squared.
    """
    epi = self.strain_rate_tensor(U)
    ep_xx = epi[0,0]
    ep_yy = epi[1,1]
    ep_zz = epi[2,2]
    ep_xy = epi[0,1]
    ep_xz = epi[0,2]
    ep_yz = epi[1,2]

    # Second invariant of the strain rate tensor squared
    epsdot = 0.5 * (ep_xx**2 + ep_yy**2 + ep_zz**2) \
+ ep_xy**2 + ep_xz**2 + ep_yz**2
    return epsdot

def default_solve_params(self):
    """
    Returns a set of default solver parameters that yield good performance
    """
    nparams = {'newton_solver' :
        {
            'linear_solver' : 'cg',
            'preconditioner' : 'hybre_amg',
            'relative_tolerance' : 1e-5,
            'relaxation_parameter' : 0.7,
            'maximum_iterations' : 25,
            'error_on_nonconvergence' : False,
            'krylov_solver' :
            {
                'monitor_convergence' : False,
                'preconditioner' :
                {
                    'structure' : 'same'
                }
            }
        }
    }
    m_params = {'solver' : nparams,
                'solve_vert_velocity' : True,
                'solve_pressure' : True,
                'vert_solve_method' : 'mumps'}
    return m_params

def solve_vert_velocity(self, annotate=annotate):
    """
    Solve for vertical velocity w.
    """
    s = " ::: solving Dukowicz reduced vertical velocity ::: "
    print_text(s, self.color())

    aw = assemble(lhs(self.w_F))
    Lw = assemble(rhs(self.w_F))

    w_solver = LUSolver(self.solve_params['vert_solve_method'])
    w_solver.solve(aw, self.w.vector(), Lw, annotate=annotate)

def solve(self, annotate=False):
    """
    Perform the Newton solve of the reduced full-Stokes equations
    # zero out self.velocity for good convergence for any subsequent solves,
    # e.g. model.L_curve() :
    model.assign_variable(self.get_U(), DOLFIN_EPS, cls=self)

    def cb_ftn():
        self.solve_vert_velocity(annotate)
```

```
# compute solution :
model.home_rolled_newton_method(self.mom_F, self.U, self.mom_Jac,
                                self.mom_bcs, atol=1e-6, rtol=rtol,
                                relaxation_param=alpha, max_iter=maxit,
                                method=lin_slv, preconditioner=precon,
                                cb_ftn=cb_ftn)

u, v = self.U.split()
```

9.5 Mass loss due to basal melting

The mass loss due to melt-water flowing from the base of the ice due to internal and external friction has the effect of lowering the ice-sheet surface. In terms of velocity, the water discharge from the ice F_b – in units of meters of ice equivalent per second – transforms impenetrability conditions (9.4), (9.44) and (9.49b) to

$$\mathbf{u} \cdot \mathbf{n} = F_b \quad \text{on } \Gamma_B \quad (9.64)$$

$$\mathbf{u}_p \cdot \mathbf{n}_p = F_b \quad \text{on } \Gamma_B \quad (9.65)$$

$$\mathbf{u}_w \cdot \mathbf{n} = F_b \quad \text{on } \Gamma_B. \quad (9.66)$$

Furthermore, because the ice velocity may no longer be tangential to the basal surface, basal-traction conditions (9.3), (9.43) and (9.57) are transformed to

$$(\sigma \cdot \mathbf{n})_{\parallel} = -\beta \mathbf{u}_{\parallel} \quad \text{on } \Gamma_G \quad (9.67)$$

$$(\sigma_{PS} \cdot \mathbf{n}_p)_{\parallel} = -\beta \mathbf{u}_{p\parallel} \quad \text{on } \Gamma_G \quad (9.68)$$

$$(\sigma_{RS} \cdot \mathbf{n})_{\parallel} = -\beta \mathbf{u}_{w\parallel} \quad \text{on } \Gamma_G. \quad (9.69)$$

Reformulation of the full-Stokes variational principle of §9.1 utilizing basal-melt-adjusted boundary conditions (9.64, 9.67) leads to

$$\begin{aligned} \mathcal{A}(\mathbf{u}, p) = & + \int_{\Omega} (V(\dot{\epsilon}_e^2) - \rho \mathbf{g} \cdot \mathbf{u} - p \nabla \cdot \mathbf{u}) \, d\Omega \\ & + \int_{\Gamma_B} \left(\Lambda(\mathbf{u} \cdot \mathbf{n} - F_b) + \frac{1}{2} \beta \mathbf{u}_{\parallel} \cdot \mathbf{u}_{\parallel} \right) \, d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n} \cdot \mathbf{u} \, d\Gamma_L, \end{aligned} \quad (9.70)$$

where $\mathbf{u}_{\parallel} = \mathbf{u} - (\mathbf{u} \cdot \mathbf{n})\mathbf{n}$. Reformulation of the plane-strain variational principle of §9.3 utilizing basal-melt-adjusted boundary conditions (9.65, 9.68) leads to

$$\begin{aligned} \mathcal{A}_{PS}(\mathbf{u}_p, p) = & + \int_{\Omega} (V(\dot{\epsilon}_{PS}^2) - \rho \mathbf{g}_p \cdot \mathbf{u}_p - p \nabla \cdot \mathbf{u}_p) \, d\Omega \\ & + \int_{\Gamma_B} \left(\Lambda_{PS}(\mathbf{u}_p \cdot \mathbf{n}_p - F_b) + \frac{1}{2} \beta \mathbf{u}_{p\parallel} \cdot \mathbf{u}_{p\parallel} \right) \, d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n}_p \cdot \mathbf{u}_p \, d\Gamma_L, \end{aligned} \quad (9.71)$$

where $\mathbf{u}_{p\parallel} = \mathbf{u}_p - (\mathbf{u}_p \cdot \mathbf{n}_p)\mathbf{n}_p$.

Closer examination of melt-adjusted impenetrability condition (9.64),

$$u n_x + v n_y + w n_z = F_b \quad \text{on } \Gamma_B,$$

suggests that given $n_z \neq 0$, the melt-adjusted basal vertical velocity is given by

$$w(B) = \frac{F_b - u(B)n_x - v(B)n_y}{n_z}. \quad (9.72)$$

This expression is then be used in place of Equation (9.28) to solve first-order vertical-velocity-component relation (9.27). Finally, reformulation of the reformulated-Stokes variational principle of §9.4 utilizing basal-melt-adjusted boundary conditions (9.66, 9.69) leads to

$$\begin{aligned} \mathcal{A}_{RS}(\mathbf{u}_w, p) = & + \int_{\Omega} (V(\dot{\epsilon}_{RS}^2) - \rho \mathbf{g} \mathbf{u}_h \cdot (\nabla S)_h) \, d\Omega \\ & + \int_{\Gamma_B} \frac{1}{2} \beta \left(\mathbf{u}_h \cdot \mathbf{u}_h + \left(\frac{F_b - \mathbf{u}_h \cdot \mathbf{n}_h}{n_z} \right)^2 \right) \, d\Gamma_B \\ & + \int_{\Gamma_L} f_w \mathbf{n} \cdot \mathbf{u}_w \, d\Gamma_L, \end{aligned} \quad (9.73)$$

and reformulated variational problem (9.50) governing the solution of w^h ,

$$+ \int_{\Omega} \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w^h}{\partial z} \right) \chi \, d\Omega \quad (9.74)$$

$$+ \int_{\Gamma_B} (u n_x + v n_y + w^h n_z - F_b) \chi \, d\Gamma_B = 0. \quad (9.75)$$

These are the forms of the action principles solved by CSLVR, demonstrated by Code Listings 9.2, 9.3, 9.4, and 9.5.

9.6 Stokes variational forms

Each of momentum model equations (9.13), (9.32), (9.48), and (9.63) is nonlinear due to the velocity dependence of the viscosity η , and as such cannot be solved directly. Instead, we determine the k unknown field variables by solving for the direction of decent of these actions using the method described in §6.1. This method forms the Gâteaux derivative of each of model extremums (9.13), (9.32), (9.48), and (9.63) with respect to a test function.

The variational problem associated with full-Stokes action extremum (9.13), first-order action extremum (9.32), reformulated-Stokes action extremum (9.63), and plane-strain action extremum (9.48) consists of finding (see trial space (1.10)) $\mathbf{U} = [u \ v \ w \ p]^T \in \mathbf{S}_E^h \subset (\mathcal{H}^1(\Omega))^4$, $\mathbf{U}_h = [u \ v]^T \in \mathbf{S}_E^h \subset (\mathcal{H}^1(\Omega))^2$, or $\mathbf{U}_p = [u \ w \ p]^T \in \mathbf{S}_E^h \subset (\mathcal{H}^1(\Omega))^3$ such that

$$\lim_{\epsilon \rightarrow 0} \left\{ \frac{\delta}{\delta \mathbf{U}} \mathcal{A}(\mathbf{U} + \epsilon \Phi) \right\} = 0 \quad (9.76)$$

$$\lim_{\epsilon \rightarrow 0} \left\{ \frac{\delta}{\delta \mathbf{U}_h} \mathcal{A}_{BP}(\mathbf{U}_h + \epsilon \Phi_h) \right\} = 0 \quad (9.77)$$

$$\lim_{\epsilon \rightarrow 0} \left\{ \frac{\delta}{\delta \mathbf{U}_h} \mathcal{A}_{RS}(\mathbf{U}_h + \epsilon \Phi_h) \right\} = 0 \quad (9.78)$$

$$\lim_{\epsilon \rightarrow 0} \left\{ \frac{\delta}{\delta \mathbf{U}_p} \mathcal{A}_{BP}(\mathbf{U}_p + \epsilon \Phi_p) \right\} = 0 \quad (9.79)$$

for all test functions (see test space (1.11)) $\Phi \in \mathbf{S}_0^h \subset (\mathcal{H}^1(\Omega))^4$, $\Phi_h \in \mathbf{S}_0^h \subset (\mathcal{H}^1(\Omega))^2$, and $\Phi_p \in \mathbf{S}_0^h \subset (\mathcal{H}^1(\Omega))^3$.

Note that when using the method described in §6.1 to solve variational forms (9.76), (9.77), (9.78), and (9.79), \mathbf{U} , \mathbf{U}_h , and \mathbf{U}_p are not trial functions, but are instead containers for the current solution approximation. In this case trial functions

Table 9.1: Variable values for ISMIP-HOM simulations.

Variable	Value	Units	Description
$\dot{\epsilon}_0$	10^{-15}	a^{-1}	strain regularization
β	1000	$\text{kg m}^{-2} \text{a}^{-1}$	basal friction coef.
A	10^{-16}	$\text{Pa}^{-3} \text{a}^{-1}$	flow-rate factor
F_b	0	m a^{-1}	basal water discharge
a	0.5	$^\circ$	surface gradient mag.
\bar{B}	1000	m	average basal depth
b	500	m	basal height amp.
k_x	15	—	number of x divisions
k_y	15	—	number of y divisions
k_z	5	—	number of z divisions
N_e	6750	—	number of cells
N_n	1536	—	number of vertices

enter into the equations as the down-gradient direction of Gâteaux derivatives (9.76), (9.77), (9.78), or (9.79). Additionally, while in the process of solving for the decent direction of the reformulated Stokes model $\delta \mathbf{U}_h \mathcal{A}_{\text{RS}}$ (9.78), discrete variational form (9.50) must be solved for each horizontal velocity approximation \mathbf{U}_h .

9.7 ISMIP-HOM test simulations

A suitable test for the three-dimensional models defined by §9.1, §9.2, and §9.4 is the higher-order-ice-sheet-model-intercomparison project presented by Pattyn et al. (2008). This test is defined over the domain $\Omega \in [0, \ell] \times [0, \ell] \times [B, S] \subset \mathbb{R}^3$ with $k_x \times k_y \times k_z$ node discretization, and specifies the use of a surface height with uniform slope $\|\nabla S\| = a$

$$S(x) = -x \tan(a),$$

and the sinusoidally-varying basal topography

$$B(x, y) = S(x) - \bar{B} + b \sin\left(\frac{2\pi}{\ell}x\right) \sin\left(\frac{2\pi}{\ell}y\right),$$

with average basal depth \bar{B} , and basal height amplitude b (Figure 9.1). To enforce continuity, the periodic \mathbf{u}, p boundary conditions

$$\begin{aligned} \mathbf{u}(0, 0) &= \mathbf{u}(\ell, \ell) & p(0, 0) &= p(\ell, \ell) \\ \mathbf{u}(0, \ell) &= \mathbf{u}(\ell, 0) & p(0, \ell) &= p(\ell, 0) \\ \mathbf{u}(x, 0) &= \mathbf{u}(x, \ell) & p(x, 0) &= p(x, \ell) \\ \mathbf{u}(0, y) &= \mathbf{u}(\ell, y) & p(0, y) &= p(\ell, y) \end{aligned}$$

were used. Lastly, the basal traction coefficient is set to $\beta = 1000$ which has the effect of creating a no-slip boundary condition along the basal surface, while $A = 10^{-16}$ is used as an isothermal rate factor for viscosity η . Table 9.1 lists these coefficients and values, and the CSLVR script used to solve this problem is shown in Code Listing 9.7.

The applicability of each of the §9.1, §9.2, and §9.4 momentum models for given basal gradient is tested by performing

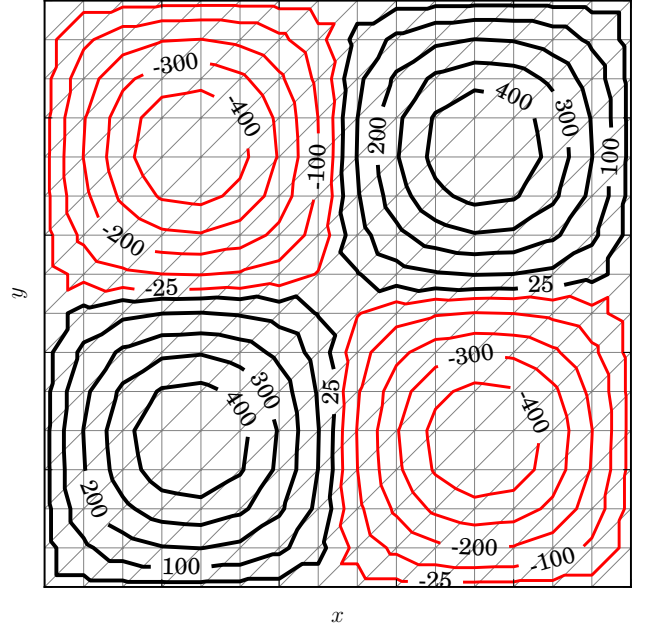


Figure 9.1: ISMIP-HOM A bedrock topography deviation from the average $\bar{B} = B + \bar{B} - S$ over the $\ell \times \ell$ square km grid. The CSLVR script used to generate the figure is shown in Code Listing 9.6.

simulations over the range of maximum domain widths $\ell = 5$ km, 8 km, 10 km, and 15 km. Results indicate that the velocity approximated by the first-order and reformulated-Stokes models approach the full-Stokes solution as ℓ increases (Figure 9.2). The reformulated-Stokes model produces a quantitatively closer result to the full-Stokes model than the first-order model for all of the experiments performed.

It was also interesting to observe that the velocity divergence $\nabla \cdot \mathbf{u}$ was largest near the bed, and is most likely due to the fact that inclusion of the impenetrability constraint $\mathbf{u} \cdot \mathbf{n} = 0$ demands more from the velocity approximation than was possible from the model formulations. Similar to the surface velocity, there was observed significant variation of basal velocity divergence between each model (Figure 9.3), with decreasing observed variance as ℓ increases.

9.8 Plane-strain simulation

For an example simulation of the plane-strain model of §9.3, we create a two-dimensional ice-sheet with surface height

$$S(x) = \left(\frac{H_{\max} + B_0 - S_0}{2} \right) \cos\left(\frac{2\pi}{\ell}x\right) + \left(\frac{H_{\max} + B_0 + S_0}{2} \right),$$

with thickness at the divide H_{\max} , height of terminus above water S_0 , depth of ice terminus below water B_0 , and width ℓ .

We prescribe the sinusoidally-varying basal topography

$$B(x) = b \cos\left(n_b \frac{2\pi}{\ell} x\right) + B_0,$$

with amplitude b and number of bumps n_b . The basal traction field used followed the same sinusoidal variation as the surface,

$$\beta(x) = \left(\frac{\beta_{\max} - \beta_{\min}}{2}\right) \cos\left(\frac{2\pi}{\ell} x\right) + \left(\frac{\beta_{\max} + \beta_{\min}}{2}\right)$$

with maximum value β_{\max} and minimum value β_{\min} . The specific values used by the simulation are listed in Table 9.2 with results depicted in Figure 9.5 generated by Code Listing 9.8.

Table 9.2: Variable values for plane-strain simulation.

Variable	Value	Units	Description
$\dot{\epsilon}_0$	10^{-15}	a^{-1}	strain regularization
A	10^{-16}	$\text{Pa}^{-3} \text{a}^{-1}$	flow-rate factor
F_b	0	m a^{-1}	basal water discharge
k_x	150	—	number of x divisions
k_z	10	—	number of z divisions
N_e	3000	—	number of cells
N_n	1661	—	number of vertices
ℓ	400	km	width of domain
H_{\max}	4000	m	thickness at divide
S_0	100	m	terminus height
B_0	-200	m	terminus depth
n_b	25	—	number of bed bumps
b	50	m	bed bump amplitude
β_{\max}	50	$\text{kg m}^{-2} \text{a}^{-1}$	max basal traction
β_{\min}	0.2	$\text{kg m}^{-2} \text{a}^{-1}$	min basal traction

Code Listing 9.6: CSLVR script used to generate ISMIP-HOM basal topography Figure 9.1.

```
from csldr import *

# create a mesh and basal geometry identical to our problem, but now
# in two dimension, cause we only want to plot the basal geometry pattern :
p1 = Point(0.0, 0.0)
p2 = Point(1.0, 1.0)
mesh = RectangleMesh(p1, p2, 15, 15)

# where we're going to save that sweet plot :
plt_dir = './.../images/momentum/ISMIP_HOM_A/'

# only need a 2D-model for this task, nevermind periodic boundaries too :
model = D2Model(mesh, out_dir = plt_dir)
bed = Expression('500.0 * sin(2*pi*x[0]/L) * sin(2*pi*x[1]/L)',
                 L=1, element=model.Q.ufl_element())

# initialize the one thing we care about (we could do other stuff too) :
model.init_B(bed)

# figure out the levels and plot them
B_min = model.B.vector().min() # we know this, but let's be sure
B_max = model.B.vector().max() # we know this, but let's be sure
B_lvls = array([B_min, -400, -300, -200, -100, -25,
                25, 100, 200, 300, 400, B_max])

# this time, let's plot the topography like a topographic map :
plot_variable(u = model.B, name = 'B', direc = plt_dir,
              figsize = (5,5), levels = B_lvls, tp = True,
              show = False, cb = False, contour_type = 'lines',
              hide_ax_tick_labels = True)
```

Code Listing 9.7: CSLVR script which solves the ISMIP-HOM experiment of §9.7.

```
from csldr import *

a = 0.5 * pi / 180 # surface slope in radians
L = 5000 # width of domain (also 8000, 10000, 14000)

# create a generic box mesh, we'll fit it to geometry below :
```

```
p1 = Point(0.0, 0.0, 0.0) # origin
p2 = Point(L, L, 1) # x, y, z corner
mesh = BoxMesh(p1, p2, 15, 15, 5) # a box to fill the void

# output directories :
out_dir = './ISMIP_HOM_A_results/BP/'
plt_dir = './.../images/momentum/ISMIP_HOM_A/xsmall/BP/'

# we have a three-dimensional problem here, with periodic lateral boundaries :
model = D3Model(mesh, out_dir = out_dir, use_periodic = True)

# the ISMIP-HOM experiment A geometry :
surface = Expression('x[0] * tan(a)', a=a,
                    element=model.Q.ufl_element())
bed = Expression('x[0] * tan(a) - 1000.0 + 500.0 * \
                + sin(2*pi*x[0]/L) * sin(2*pi*x[1]/L)',
                a=a, L=L, element=model.Q.ufl_element())

# mark the exterior facets and interior cells appropriately :
model.calculate_boundaries()

# deform the mesh to match our desired geometry :
model.deform_mesh_to_geometry(surface, bed)

# initialize all the pertinent variables :
model.init_beta(1000) # really high friction
model.init_A(1e-16) # cold, isothermal rate-factor

# we can choose any of these to solve our 3D-momentum problem :
mom = MomentumDukowiczBP(model)
#mom = MomentumDukowiczStokesReduced(model)
#mom = MomentumDukowiczStokes(model)
mom.solve()

# let's investigate the velocity divergence :
divU = project(div(model.U3))

# the purpose for everything below this line is data visualization :
#=====

# save these files with a name that makes sense for use with paraview :
model.save_xdmf(model.p, 'p')
model.save_xdmf(model.U3, 'U')
model.save_xdmf(divU, 'divU')

# create the bed and surface meshes :
model.form_bed_mesh()
model.form_srf_mesh()

# create 2D models :
bedmodel = D2Model(model.bedmesh, out_dir)
srfmodel = D2Model(model.srfmesh, out_dir)

# we don't have a function for this included in the 'model' instance,
# so we have to make one ourselves :
divU_b = Function(bedmodel.Q)

# function allows Lagrange interpolation between different meshes :
bedmodel.assign_submesh_variable(divU_b, divU)
srfmodel.assign_submesh_variable(srfmodel.U3, model.U3)
srfmodel.init_U_mag(srfmodel.U3) # calculates the velocity magnitude
bedmodel.assign_submesh_variable(bedmodel.p, model.p)

# figure out some nice-looking contour levels :
U_min = srfmodel.U_mag.vector().min()
U_max = srfmodel.U_mag.vector().max()
#U_lvls = array([U_min, 89, 90, 92, 94, 96, 98, 100, U_max])
U_lvls = array([68, 70, 72, 74, 76, 78, 80, 82, 84, 86, 88, 90, 92, 98])

p_min = bedmodel.p.vector().min()
p_max = bedmodel.p.vector().max()
p_lvls = array([4e6, 5e6, 6e6, 7e6, 8e6, 9e6, 1e7, 1.1e7, 1.2e7, p_max])

d_min = divU_b.vector().min()
d_max = divU_b.vector().max()
d_lvls = array([d_min, -5e-3, -2.5e-3, -1e-3,
                1e-3, 2.5e-3, 5e-3, d_max])

# these functions allow the plotting of an arbitrary FEniCS function or
# vector that reside on a two-dimensional mesh (hence the D2Model
# instantiations above.
plot_variable(u = srfmodel.U3, name = 'U_mag', direc = plt_dir,
              levels = U_lvls,
              cmap = 'viridis',
              tp = True,
              show = False,
              extend = 'min',
              cb.format = '%g')

plot_variable(u = bedmodel.p, name = 'p', direc = plt_dir,
              levels = p_lvls,
              cmap = 'viridis',
              tp = True,
              show = False,
              extend = 'min',
              cb.format = '%.1e')

plot_variable(u = divU_b, name = 'divU', direc = plt_dir,
              cmap = 'RdGy',
              levels = d_lvls,
              tp = True,
              show = False,
              extend = 'neither',
              cb.format = '%.1e')
```

Code Listing 9.8: CSLVR source code used to solve the plane-strain problem of §9.8.

```
from csldr import *

# this problem has a few more constants than the last :
l = 400000.0 # width of the domain
Hmax = 4000 # thickness at the divide
S0 = 100 # terminus height above water
B0 = 200 # terminus depth below water
nb = 25 # number of basal bumps
b = 50 # amplitude of basal bumps
betaMax = 50.0 # maximum traction coefficient
betaMin = 0.2 # minimum traction coefficient

# this time, we solve over a 2D domain, we'll deform to geometry below :
p1 = Point(-1/2, 0.0) # lower left corner
p2 = Point(1/2, 1.0) # upper right corner
nx = 150 # number of x-element divisions
nz = 10 # number of z-element divisions
mesh = RectangleMesh(p1, p2, nx, nz)

# set some output directories :
out_dir = 'ps_results/'
plt_dir = './.../images/momentum/plane_strain/'
```

```

# this is a 'lateral' model, defined in the x,z plane :
model = LatModel(mesh, out_dir = out_dir, use_periodic = False)

# the geometry and desired traction :
S = Expression('(Hmax+B0-S0)/2*cos(2*pi*x[0]/1) + (Hmax+B0+S0)/2',
               Hmax=Hmax, B0=B0, S0=S0, l=1,
               element = model.Q.ufl_element())
B = Expression('b*cos(nb*2*pi*x[0]/1) + B0',
               b=b, l=1, B0=B0, nb=nb,
               element = model.Q.ufl_element())
b = Expression('(bMax - bMin)/2.0*cos(2*pi*x[0]/1) + (bMax + bMin)/2.0',
               bMax=betaMax, bMin=betaMin, l=1,
               element = model.Q.ufl_element())

# deform the mesh, just like we did with the 3D model :
model.deform_mesh_to_geometry(S, B)

# calculate the boundaries for proper variational-form integration :
model.calculate_boundaries(mask=None)

# initialize the constants that we want, here like the ISMIP-HOM exp. :
model.init_beta(b) # traction
model.init_A(1e-16) # flow-rate factor

# only one type of momentum physics for this problem :
mom = MomentumDukowiczPlaneStrain(model)
mom.solve()

# plotting :
#####

# let's calculate the velocity speed :
model.init_U_mag(model.U3)

U_min = model.U_mag.vector().min()
U_max = model.U_mag.vector().max()
U_lvls = array([U_min, 5e3, 1e4, 2e4, 3e4, 4e4, 5e4, U_max])

p_min = model.p.vector().min()
p_max = model.p.vector().max()
p_lvls = array([p_min, 5e3, 1e7, 1.5e7, 2e7, 2.5e7, 3e7, 3.5e7, p_max])

beta_min = model.beta.vector().min()
beta_max = model.beta.vector().max()
beta_lvls = array([beta_min, 5.0, 10, 15, 20, 25, 30, 35, 40, 45, beta_max])

plot_variable(u = model.U_mag, name = 'U_mag', direc = plt_dir,
              figsize = (8,3),
              title = r'$\textbf{u}$',
              cmap = 'viridis',
              levels = U_lvls,
              tp = True,
              show = False,
              ylabel = r'$z$',
              equal_axes = False,
              cb_format = '%.1e')

plot_variable(u = model.p, name = 'p', direc = plt_dir,
              figsize = (8,3),
              title = r'$p$',
              cmap = 'viridis',
              levels = p_lvls,
              tp = True,
              show = False,
              ylabel = r'$z$',
              equal_axes = False,
              cb_format = '%.1e')

plot_variable(u = model.beta, name = 'beta', direc = plt_dir,
              figsize = (8,3),
              title = r'$\beta$',
              cmap = 'viridis',
              levels = beta_lvls,
              tp = True,
              show = False,
              ylabel = r'$z$',
              equal_axes = False,
              cb_format = '%g')

```

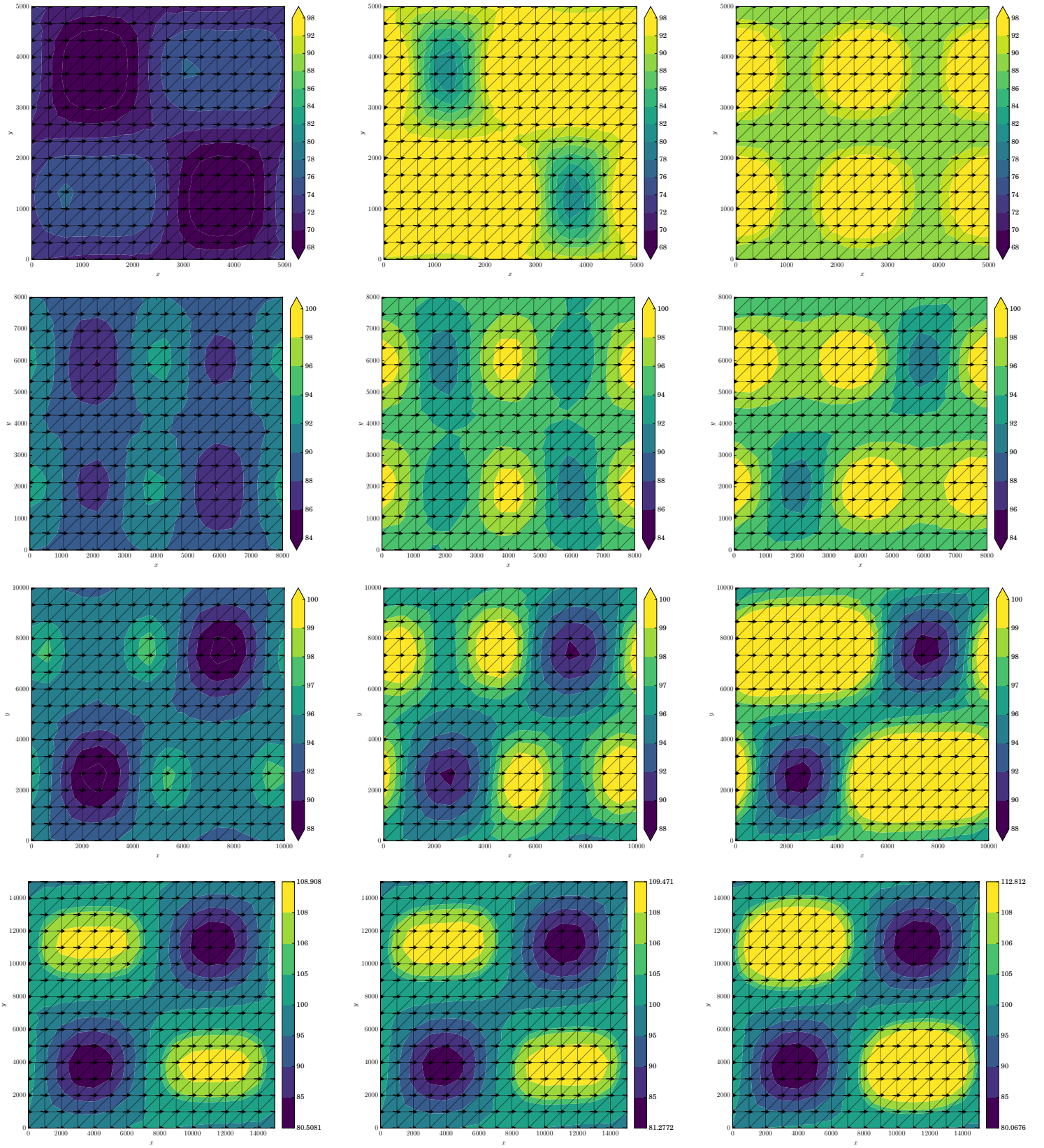


Figure 9.2: Surface velocity u_s for the ISMIP-HOM test experiment using a 5×5 square km grid (first row), 8×8 km² grid (second row), 10×10 km² grid (third row), and 15×15 km² grid (fourth row). The first column are results attained using the full-Stokes model from §9.1, the second column using the first-order model from §9.2, and the third column was generated using the reformulated-Stokes model of §9.4.

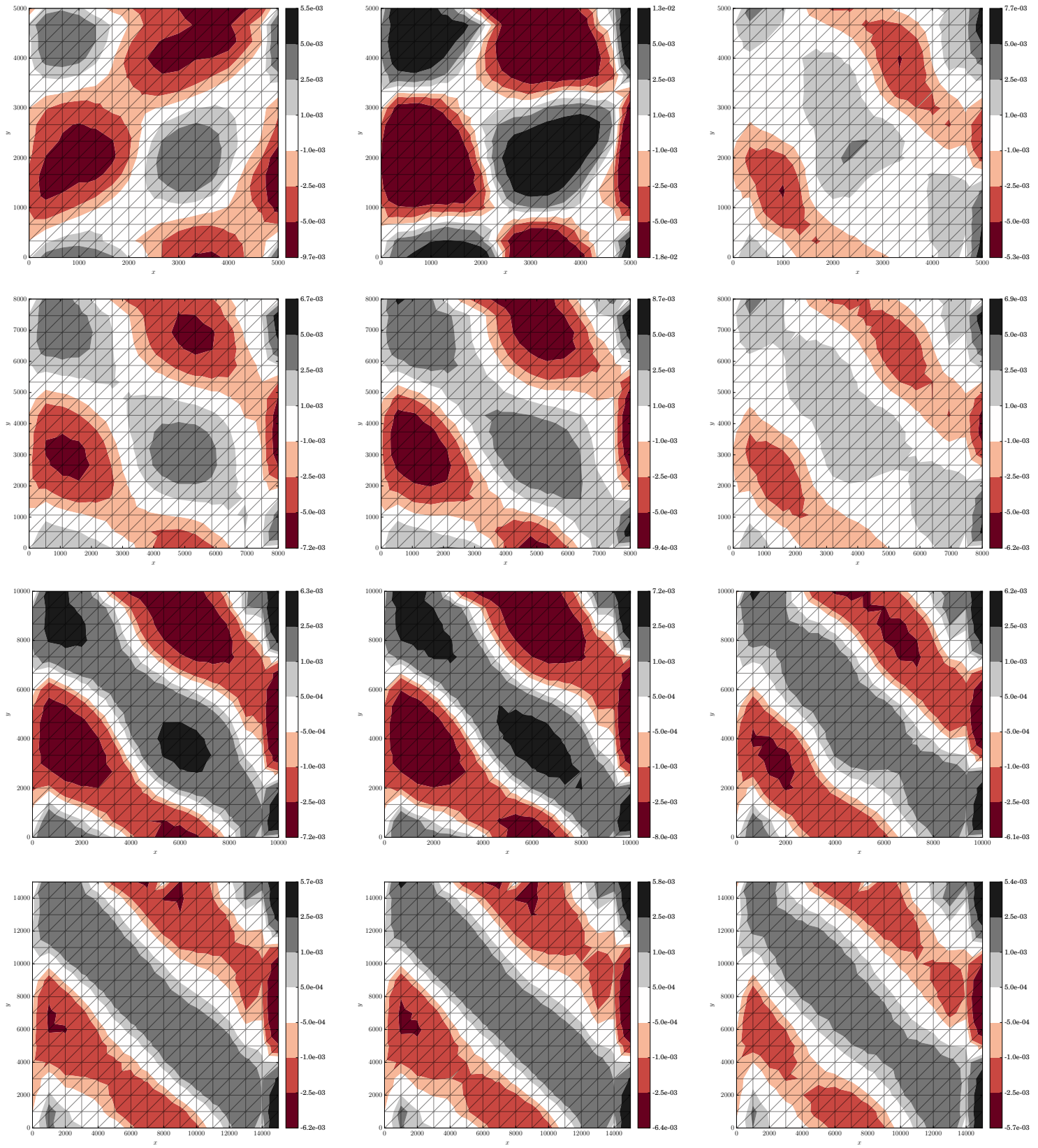


Figure 9.3: Basal velocity divergence $\nabla \cdot \mathbf{u}|_B$ for the ISMIP-HOM test experiment using a 5×5 square km grid (first row), 8×8 km² grid (second row), 10×10 km² grid (third row), and 15×15 km² grid (fourth row). The first column are results attained using the full-Stokes model from §9.1, the second column using the first-order model from §9.2, and the third column was generated using the reformulated-Stokes model of §9.4.

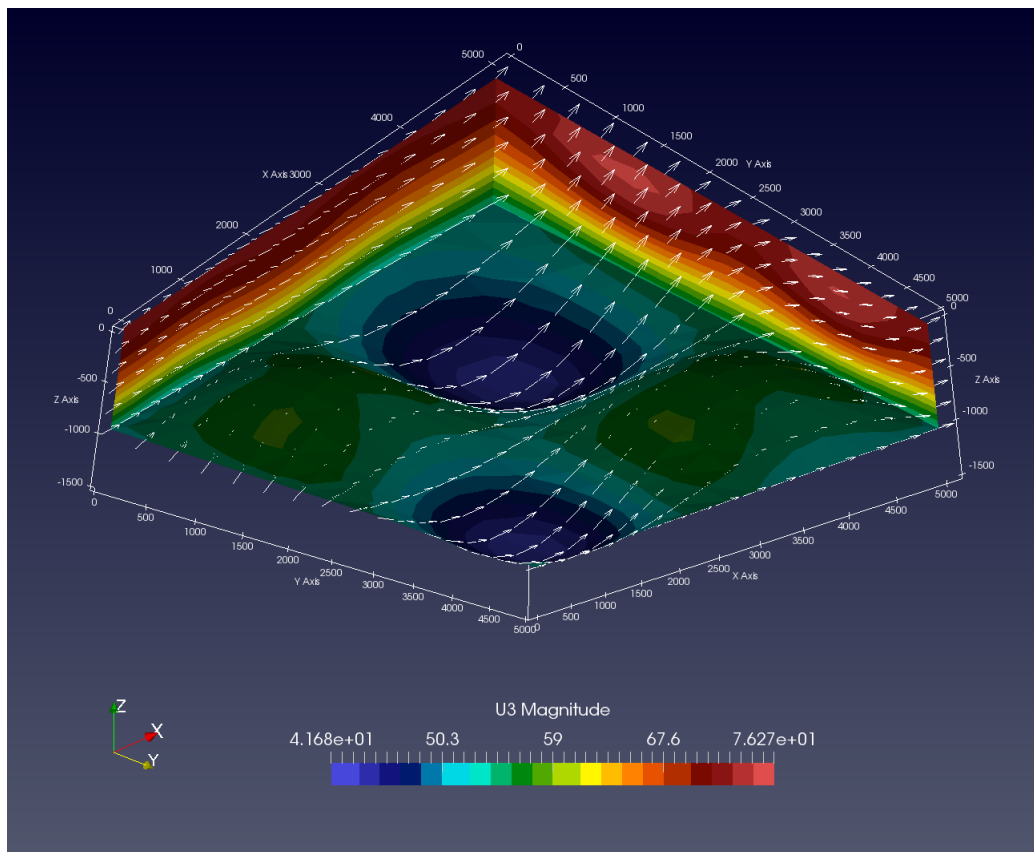
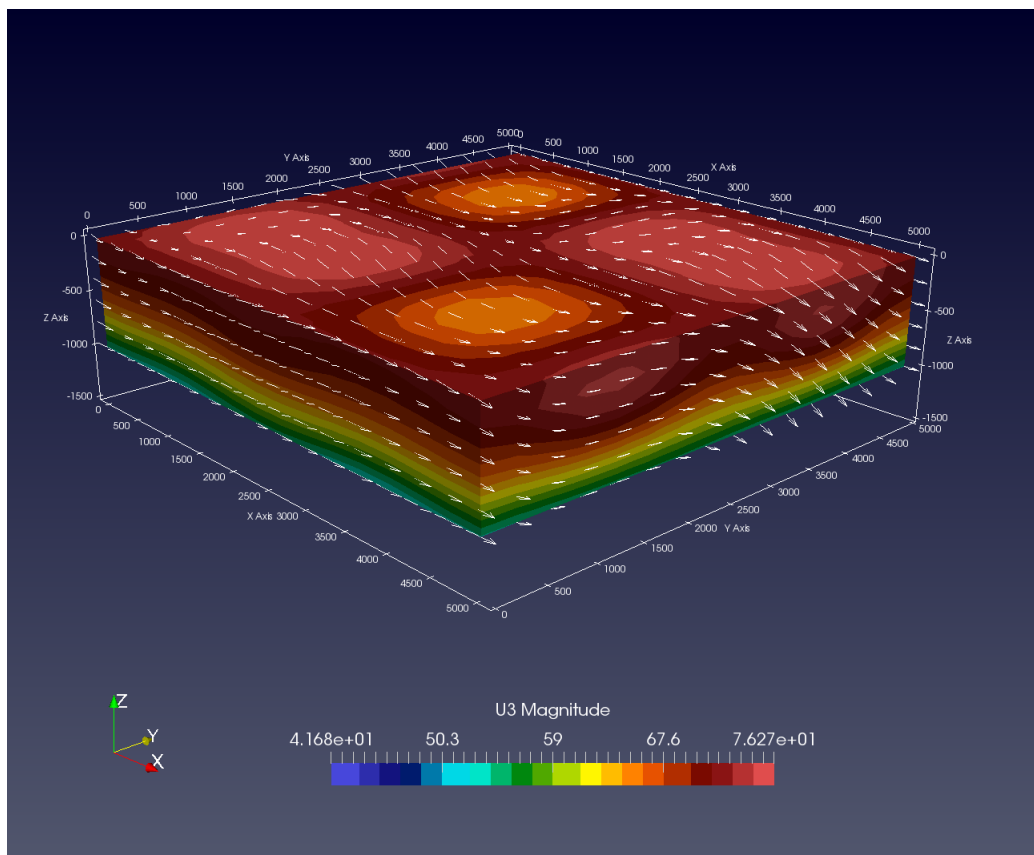


Figure 9.4: Full-Stokes velocity \mathbf{u} view from above (top) and below (bottom) with a domain of 5×5 square km.

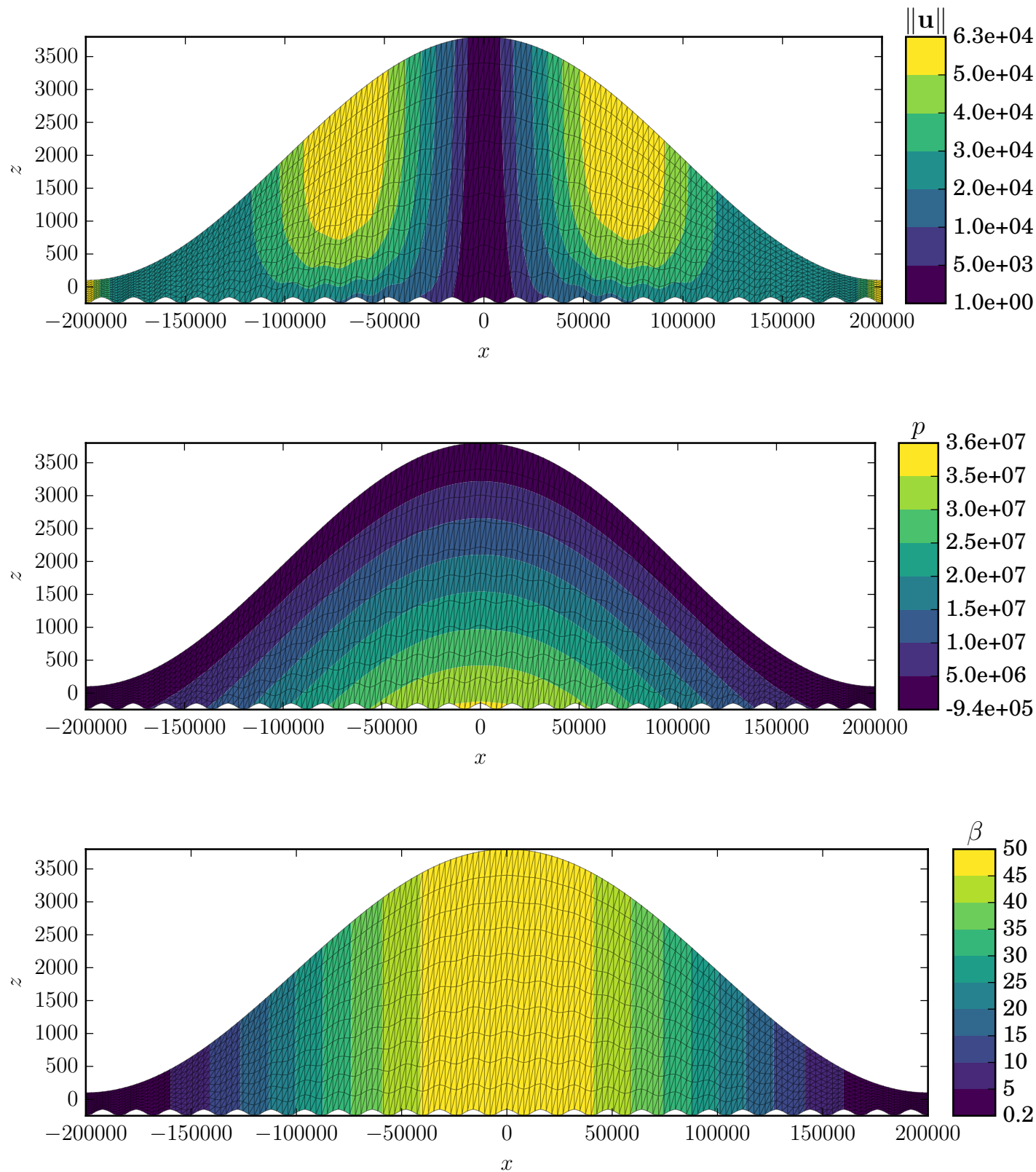


Figure 9.5: The plane-strain profile results for the simulation outlined by §9.8; velocity magnitude $\|\mathbf{u}\|$ (top), pressure p (middle), and prescribed basal traction β (bottom).

Chapter 10

Internal energy balance

The internal energy boundary conditions appropriate to the base of an ice-sheet or glacier are complicated by the fact that both essential and natural types have been specified. Proposed here is a unification of these conditions in a single natural form, presented as a water content minimization problem whereby an observed maximum value of moisture retention within ice is enforced. Using this method, previous constraints on the basal energy flux are no longer required, allowing abnormally high intra-ice water contents resulting from internal friction to be drained from the ice using established energy transport equations pertinent to polythermal glaciers. An algorithm is presented in §12.2 which couples this method with a surface-velocity data-assimilation procedure for basal traction (described in §12.1), resulting in a set fully thermomechanically coupled basal traction, velocity, internal energy, and basal water discharge.

10.1 Introduction

Within a polythermal glacier or ice-sheet, both liquid and solid phases are present. It follows that any mathematical description of polythermal ice must account for the role of liquid water in terms of both rheology and energy. First, ice is defined as cold if a change in energy leads to a change in temperature alone, and temperate if a change in energy leads to a change in water content alone. Methods from mixture theory have been incorporated and models proposed which either track the transition surface from cold to temperate (CTS) explicitly (Hutter, 1982; Greve, 1997; Greve and Blatter, 2009; Blatter and Greve, 2015), or solve the internal energy as a single continuous variable (Aschwanden and Blatter, 2009; Aschwanden et al., 2012). Irrespective of the implementation, these models yield a potentially time-varying distribution of internal energy, which via bijection provides temperature and water content.

For cold ice, defined as ice with zero liquid water content, an advection-diffusion equation with a strain-heating source describes heat flow (Paterson, 1994). The energy distribution in temperate ice is more complex. Instead of raising the temperature of ice, strain-heating in these areas generates water. This water, once produced, is advected along the trajectory of ice flow. Additionally, the water is thought to either diffuse in the same way as heat (Hutter, 1982), as considered here, or move

in a similar fashion to groundwater in a Darcy-type pattern whereby pockets of water drain under pressure through micro-connecting veins (Fowler, 1982). The precise mechanism governing the non-advective transport of water remains unclear.

In order to solve equations governing the transport of energy defining these models, appropriate boundary conditions must be prescribed. Over the surface in contact with atmosphere, estimated water content and temperatures are readily applied as essential-type conditions. Over the basal surface in contact with bedrock, both geothermal and frictional heat is presented to the ice as a natural-type boundary. Further complexity arises in imposing this natural boundary condition.

For cold ice, the entirety of geothermal and frictional heat flows into the interior and raises the temperature. The temperature of ice becomes fixed once it reaches its pressure-melting point, and the energy flux into the ice then becomes proportional to its basal melting rate adjusted by water discharge (Greve, 1997). Additionally, for interior ice with temperature at its pressure-melting point and possessing a non-zero strain-heat source, a *temperate zone* is created. While there is no reason a temperate zone could not be formed within the ice interior, the strain-rate is greatest near the bed for ice-sheets and Canadian-type glaciers considered here (Aschwanden et al., 2012), and thus temperate zones typically lie near the basal surface at these localities (Figure 8.1).

As there is no existing constitutive relation explaining the water flux either out of or into the ice in temperate zones, the flux of energy into the ice has been previously specified to be in balance with the flux of water out of the ice, such that a homogeneous natural boundary condition exists (Greve, 1997; Greve and Blatter, 2009; Aschwanden and Blatter, 2009; Aschwanden et al., 2012; Kleiner et al., 2015; Blatter and Greve, 2015). In the process of applying this homogeneous energy boundary condition to observed glacial geometries and surface boundary conditions, complications arise when strain-heating creates unreasonably high water content within the ice. To address this issue, models heretofore have either moved all internally-generated water above a threshold directly into a basal-water storage-layer (Greve, 1997), or eliminated some fraction of water at a rate proportional to its magnitude (Aschwanden et al., 2012, section 4.6). In so doing, these models abandon their mathematical formulations in order to ensure that the ice does not deviate from an expected maximum wa-

ter content.

The work presented here addresses this shortcoming by expressing the basal boundary condition as a control optimization problem (Bryson and Ho, 1975; Nosedal and Wright, 2000), whereby an observed maximum basal water content is enforced. This method is coupled with the basal traction inversion procedure outlined by MacAyeal (1993) in order to incorporate satellite- and radar-observed ice surface velocities. The result of this optimization process is consistent set of energy, basal water discharge, velocity, pressure, and basal traction fields. The method is applicable to all diffusive-type energy balance models, requiring only the re-specification of the basal boundary condition and that a method of water transport is available; i.e. a non-zero non-advective water flux coefficient is assumed.

10.2 Mathematical foundation

To begin, the internal energy of ice is defined from the *constitutive equation for internal energy*, as found in Greve and Blatter (2009),

$$\theta(T, W) = \int_{T_0}^T c_i(T') dT' + WL_f, \quad (10.1)$$

with absolute temperature T , reference temperature T_0 , and water content or moisture density $W \in [0, 1]$. Water content W is defined as the ratio of the mass of water contained within a unit-volume of water-ice mixture to the mass of the mixture, such that $W = 0$ and $W = 1$ corresponds with 100% ice and 100% water, respectively. Energy definition (10.1) is in turn characterized by the definitions of *sensible* heat capacity c_i and *latent* heat capacity L_f ; the amount of energy required to raise one unit mass of ice one unit of temperature and completely melt one unit mass of ice, respectively.

The *energy balance* takes the form as presented by Greve (1997),

$$\rho \frac{d\theta}{dt} = -\nabla \cdot \mathbf{q} + Q, \quad (10.2)$$

with mixture density ρ , strain-heat Q , and energy-flux composed of sensible and latent terms

$$\mathbf{q} = \mathbf{q}_s + \mathbf{q}_l, \quad (10.3)$$

derived from *Fourier's Law* of conduction (Davis, 2013), with thermal conductivity k associated with temperature T and non-advective water flux coefficient v associated with water content W ,

$$\mathbf{q}_s = -k(\theta)\nabla T \quad \mathbf{q}_l = -v(\theta)\nabla W. \quad (10.4)$$

Note here that the latent heat flux coefficient v differs from previous formulations such as Greve (1997) and Aschwanden et al. (2012) in that it has multiplicatively absorbed latent heat capacity L_f .

Bulk water-ice mixture properties apply to thermal conductivity k , heat capacity c , and density ρ ,

$$k = (1 - W)k_i + Wk_w \quad (10.5)$$

$$c = (1 - W)c_i + Wc_w \quad (10.6)$$

$$\rho = (1 - W)\rho_i + W\rho_w, \quad (10.7)$$

where the subscripts i and w respectively refer to ice and water. Thermal conductivity k_i in (10.5) has been shown to relate to temperature (Yen, 1981; Ritz, 1987; Greve and Blatter, 2009),

$$k_i = 9.828 \exp(-5.7 \times 10^{-3} T), \quad (10.8)$$

as well as heat capacity c_i in (10.6),

$$c_i = a + bT, \quad a = 146.3, \quad b = 7.253, \quad (10.9)$$

while densities ρ_i , ρ_w and latent heat properties k_w , c_w , and L_f are taken as constant.

As stated in Hutter (1982); Greve (1997); Greve and Blatter (2009); Aschwanden et al. (2012), because the maximum water retention of ice has mostly been observed to be less than 5% of the total mass, the maximum change in density is less than 0.5%. Hence it is reasonable to abandon separate momentum balances for disparate masses of ice and water and instead treat the mixture as a single homogeneous and incompressible fluid. Thus it is *demanded* that

$$W \leq W_c \leq 0.05, \quad (10.10)$$

where W_c is an observed maximum water content.

Using definition (10.9) for heat capacity c_i , the integral in energy definition (10.1) is evaluated using for simplicity $T_0 = 0$, providing the quadratic equation for energy

$$\theta(T, W) = aT + \frac{b}{2}T^2 + WL_f. \quad (10.11)$$

The temperature of ice is also constrained by its melting point, shown to be dependent on pressure p by the *Clausius-Clapeyron* relationship (Paterson, 1994)

$$T_m = T_w - \gamma p, \quad (10.12)$$

with triple point of water $T_w = 273.15$ and empirically-derived coefficient $\gamma = 9.8 \times 10^{-8}$ (represented by the dashed red line in Figure 10.4). Using this relation, the internal energy of pure ice raised to its pressure-melting point is

$$\theta_m = \theta(T_m, 0) = aT_m + \frac{b}{2}T_m^2, \quad (10.13)$$

while the internal energy of ice that has been $(W \times 100)\%$ melted is

$$\theta(T_m, W) = \theta_m + WL_f. \quad (10.14)$$

Note here that because $0 \leq W \leq 1$, temperatures above the pressure melting point are explicitly forbidden. This is acceptable, given the very low allowable percentage of water as

demanded by (10.10); water internal to the ice will be in close contact with ice and should thus not exceed the melting temperature of ice.

Using (10.14), water content W of the mixture is defined as the fraction of internal energy above that of pure ice at the pressure melting point to the specific latent heat of fusion L_f ,

$$W(\theta) = \begin{cases} \frac{\theta - \theta_m}{L_f}, & \theta > \theta_m \\ 0, & \theta \leq \theta_m \end{cases}, \quad (10.15)$$

while temperature T is derived using the quadratic formula with $W = 0$ in Equation (10.11),

$$T(\theta) = \begin{cases} T_m, & \theta > \theta_m \\ \frac{-a + \sqrt{a^2 + 2b\theta}}{b}, & \theta \leq \theta_m \end{cases}. \quad (10.16)$$

If a different lower integration bound T_0 in (10.1) were used to calculate (10.11) and (10.13) – say $T_0 = T_m$ at pressure-melting temperature (10.12) – a simple calculation will show that (10.1) will be negative in areas with temperature below T_m , while temperature (10.16) and water content (10.15) will produce identical values as when taking $T_0 = 0$. If a more accurate estimate of internal energy is required, one may be attained by using a heat capacity appropriate for the entire range of T from zero to T_m , as compiled by Yen (1981).

Combining energy flux definitions (10.3) and (10.4) with definitions of water content (10.15) and temperature (10.16), the flow of energy may be stated

$$\mathbf{q} = \begin{cases} \mathbf{q}_s + \mathbf{q}_l = -k\nabla T_m - \nu\nabla W, & \theta > \theta_m \\ \mathbf{q}_s + \mathbf{q}_l = -k\nabla T, & \theta \leq \theta_m \end{cases}. \quad (10.17)$$

Because no universally agreed upon constitutive relation exists between latent-energy non-advective flux coefficient ν and energy θ , the use of the Fickian-type *regularizing* choice of $\nu \approx k/k_0$ for some constant k_0 as suggested by Aschwanden and Blatter (2009) is used here. The *enthalpy-gradient* method (Pham, 1995; Aschwanden and Blatter, 2009; Aschwanden et al., 2012) simplifies energy-flux term (10.17) further by using the unified energy flux

$$\mathbf{q} = -\left(\frac{\kappa}{c}\right)\nabla\theta, \quad \kappa = \begin{cases} \frac{k}{k_0}, & \theta > \theta_m \\ k, & \theta \leq \theta_m \end{cases}. \quad (10.18)$$

Note that for enthalpy $\tilde{\theta}$, the relationship to internal energy θ , pressure p , and volume V is (Yen, 1981)

$$d\tilde{\theta} = d\theta + pdV, \quad (10.19)$$

and using the same reasoning leading to demand (10.10), for incompressible ice $dV \approx 0$, and therefore enthalpy $\tilde{\theta}$ can be taken synonymously with internal energy θ in the context of glaciers and ice-sheets.

While the enthalpy-gradient method has been shown to reproduce a nearly identical CTS as models which track the CTS explicitly (Kleiner et al., 2015), great care must be taken when discretizing the system of equations (Blatter and Greve, 2015). However, the study here only requires alteration of the basal

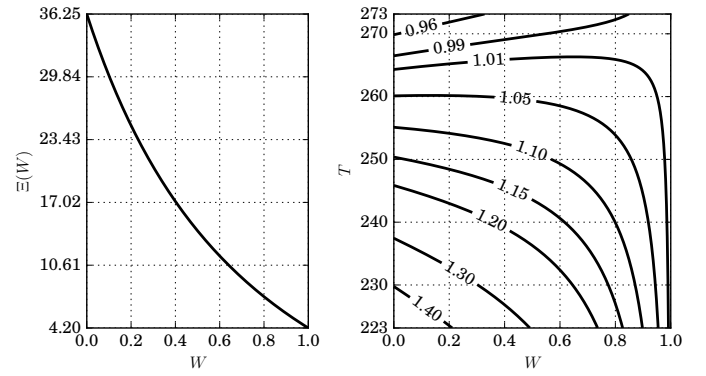


Figure 10.1: Energy diffusivity $\Xi(W, T = 268.15)$ in units of m^2a^{-1} (left) as a function of water content W only, and the ratio of $\Xi(W, T = 268.15)$ to $\Xi(W, T)$ (right) utilizing the temperature-dependent heat capacity (10.9) and thermal conductivity (10.8) in diffusivity (10.21).

boundary conditions appropriate to these models, and hence any variability in the CTS caused by either the choice of k_0 or the method of discretization of enthalpy-gradient flux (10.18) may be safely ignored, without losing generality.

The material (convective) derivative and divergence terms in energy-balance (10.2) are expanded using enthalpy-gradient flux (10.18),

$$\rho \left(\frac{\partial \theta}{\partial t} + \mathbf{u} \cdot \nabla \theta \right) = \nabla \cdot \left(\frac{\kappa}{c} \right) \cdot \nabla \theta + \frac{\kappa}{c} \nabla \cdot \nabla \theta + Q, \quad (10.20)$$

where $\mathbf{u} = [u \ v \ w]^T$ is the mixture velocity in Cartesian coordinates with respective horizontal axes (x, y) and vertical z (Figure 8.1). Note that if a constant heat capacity and thermal conductivity were used, as is commonly done, the first term on the right-hand-side of this equation would be zero and could hence be eliminated. I prefer to include this temperature relationship, as it accounts for up to 50% extra variation in diffusive capability (Figure 10.1). In the presence of water the ice mixture will be less able to conduct energy due to decreased bulk thermal conductivity (10.5), increased bulk heat capacity (10.6), and increased bulk density (10.7) used with energy-balance (10.20). This variation in diffusion is measured by dividing both sides of energy balance (10.20) by density ρ , resulting in the *diffusivity*

$$\Xi = \kappa / (\rho c), \quad (10.21)$$

so named because it is attached to the *diffusive* term $\nabla \cdot \nabla \theta$.

10.2.1 Momentum interdependence

Coupling between energy balance (10.20) and momentum balance (8.1, 8.2) occurs through pressure-melting relation (10.12, 10.13) and friction, both internally with strain-heat (8.11) and externally with basal friction heat generated by sliding over rough terrain.

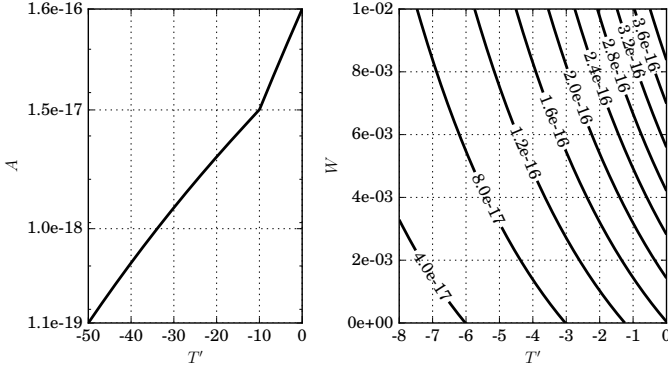


Figure 10.2: Flow rate factor (10.22) in $\text{Pa}^{-3}\text{s}^{-1}$ used in viscosity (8.10) with $W = 0$ (left), and a range of water contents (right). The change in slope in the left figure at -10°C is due to the discontinuity of parameters (10.23) and (10.24).

First, shear viscosity (8.10) and internal friction (8.11) are defined with the Arrhenius-type, energy-dependent flow-rate factor

$$A(\theta) = a_T E (1 + 181.5 W_f) \exp\left(-\frac{Q_T}{RT'}\right), \quad (10.22)$$

with enhancement factor $E = 1$ unless otherwise specified (see §12.4), universal gas constant R , empirically-constrained water content $W_f = \min\{W, 0.01\}$ (Paterson, 1994), energy-dependent flow-parameter (Patterson and Budd, 1982)

$$a_T = \begin{cases} 3.985 \times 10^{-13} & \text{s}^{-1}\text{Pa}^{-3} & T' < 263.15 \\ 1.916 \times 10^3 & \text{s}^{-1}\text{Pa}^{-3} & T' \geq 263.15 \end{cases}, \quad (10.23)$$

temperature-dependent creep activation energy

$$Q_T = \begin{cases} 6.00 \times 10^4 & \text{J mol}^{-1} & T' < 263.15 \\ 1.39 \times 10^5 & \text{J mol}^{-1} & T' \geq 263.15 \end{cases}, \quad (10.24)$$

with pressure-melting adjusted temperature $T' = T + \gamma p$ (Greve, Zwinger, and Gong, 2014). Note that rate factor (10.22) will decrease with decreasing temperature, and increase with increasing water content. This dependence on energy is thus also expressed by shear viscosity (8.10) and internal friction (8.11). Therefore, a decrease in energy θ produces stiffer ice that is more resistance to deformation, while an increase in θ produces softer ice which is easier to deform (Figure 10.2).

The second coupling between energy and momentum is by external friction heat flowing into the ice – defined analogously to internal friction (8.11) – formed from the negative product of *tangential stress* with *tangential velocity* (Greve and Blatter, 2009)

$$q_{fric} = -(\sigma \cdot \mathbf{n})_{\parallel} \cdot \mathbf{u}_{\parallel} = \beta \mathbf{u} \cdot \mathbf{u} = \beta \|\mathbf{u}\|^2, \quad (10.25)$$

where impenetrable bed condition (9.4) was used.

10.3 Energy boundary conditions

The flow of energy present on the base of the ice, when neglecting sub-glacial water transport, is a combination of geothermal q_{geo} and friction energy from sliding q_{fric} sources

$$g_N = q_{geo} + q_{fric}, \quad (10.26)$$

in units of Wm^{-2} . For cold regions this energy flux can only raise the temperature of ice, whereas for ice at its pressure melting point, the energy flux will create water along the basal surface by melting. Once generated, this water is available for transport via a sub-glacial hydraulic network. If basal water transport is prohibited, the energy flux will raise the water content along the basal surface, which may then flow under pressure through veins located between ice three-grain boundaries (Nye and Frank, 1973; Shreve, 1972; Raymond and Harrison, 1975; Llibouty, 1996). The specification of sub-glacial water transport is therefore critical for determining the correct distribution of water – and thus energy – both interior and exterior to the ice sheet. While beyond the scope of study here, basal hydraulic models may be easily incorporated into the solution method presented in the following sections.

The boundary conditions over exterior ice-sheet surface $\Gamma = \Gamma_A \cup \Gamma_W \cup \Gamma_C \cup \Gamma_T$ with atmosphere boundary Γ_A , boundary in contact with ocean Γ_W , cold or temperate basal surface without overlying temperate ice Γ_C , and temperate basal surface with overlying temperate ice Γ_T (Figure 8.1) for temperature T are

$$T = T_S \quad \text{on } \Gamma_A, \quad (10.27)$$

$$T = T_{sea} \quad \text{on } \Gamma_W, \quad (10.28)$$

$$(k \nabla T) \cdot \mathbf{n} = g_N \quad \text{on } \Gamma_C, \quad (10.29)$$

$$T = T_m \quad \text{on } \Gamma_T, \quad (10.30)$$

where seawater temperature T_{sea} may possibly be unequal to pressure melting temperature T_m . A similar set of conditions exist for water content W ,

$$W = W_S \quad \text{on } \Gamma_A, \quad (10.31)$$

$$W = W_{sea} \quad \text{on } \Gamma_W, \quad (10.32)$$

$$W = 0 \quad \text{on } \Gamma_C, \quad (10.33)$$

$$(\nu \nabla W) \cdot \mathbf{n} = \rho L_f M_b - \rho_w L_f F_b \quad \text{on } \Gamma_T, \quad (10.34)$$

with basal melting rate M_b , basal water discharge from the ice F_b , and water content on ocean boundaries W_{sea} . Note that latent energy flux (10.34) has been defined previously by Greve (1997); make the substitution $\rho_w^+ = \rho_w$, $\mathcal{P}_b^w = \rho M_b$, and $\dot{m}_b^w = \rho_w F_b$ in Equation (2.44) and $\omega^- \approx 0$ in Equation (2.51) of this work (see Appendix A).

The relationship between basal water content and the above stated basal boundary conditions may be put into perspective by considering the *basal energy balance*, defined as a combination of energy flowing into the mixture, sensible energy flux out of the mixture, and energy fluctuations caused by latent heat of fusion transitions (Greve and Blatter, 2009, section

9.3.4):

$$M_b L_f \rho = q_{geo} + q_{fric} - (k \nabla T) \cdot \mathbf{n} \quad \text{on } \Gamma_G, \quad (10.35)$$

where $\Gamma_G = \Gamma_C \cup \Gamma_T$ is the entire grounded basal surface. Note that because L_f and ρ are both positive and non-zero, if $M_b > 0$, mass is able to be accumulated by the basal hydraulic network in the form of melting ice. Likewise, if $M_b < 0$, the ice is able to accumulate mass on its basal surface in the form of freezing water, if available from the hydraulic network. Furthermore, note that for $T|_{\Gamma_G} < T_m$, the flux of temperature from the ice (10.29) inserted into basal energy balance (10.35) implies that $M_b = 0$. Finally, at temperate basal regions, essential temperature condition (10.30) applies and the basal melt rate becomes quantifiable from basal energy balance (10.35),

$$M_b = \frac{q_{geo} + q_{fric} - (k \nabla T) \cdot \mathbf{n}}{L_f \rho} \quad \text{on } \Gamma_{CT}. \quad (10.36)$$

where $\Gamma_{CT} = (\Gamma_C \cap \Gamma_T) \cup \Gamma_T$ is the entire temperate basal surface. Therefore, basal melt rate M_b is a means to quantify both the interaction of the ice with sub-glacial water by way of basal melting and accretion by freezing (referred to as *basal freeze-on*), and the flux of water into the base of the ice.

Similarly, solving for basal water discharge F_b in latent energy flux (10.34) results in

$$F_b = \frac{\rho L_f M_b - (v \nabla W) \cdot \mathbf{n}}{L_f \rho_w} \quad \text{on } \Gamma_T. \quad (10.37)$$

This is the total rate of water flowing from the ice in units of m s^{-1} . Note that if $F_b = 0$ no amount of water is able to flow from the ice. In this case, latent energy flux (10.34) is solely determined by basal melt rate (10.36) and all basally-generated melt water is available for transport to the interior of the ice as governed by enthalpy gradient flux (10.18). Basal water discharge F_b may also be negative, corresponding with water flowing into the ice from the basal hydraulic network, and positive, corresponding with water flowing out; this will respectively increase and decrease latent energy flux (10.34). For the purposes of this paper, water is not allowed to flow into the interior of the ice from the basal hydraulic network, corresponding to the requirement $F_b \geq 0$.

Next, temperature boundary conditions (10.27, 10.28) and water boundary conditions (10.31, 10.32) are combined using energy constitutive relation (10.1),

$$\theta = \int_{T_0}^{T_S} c_i(T') dT' + W_S L_f \quad \text{on } \Gamma_A \quad (10.38)$$

$$\theta = \int_{T_0}^{T_{sea}} c_i(T') dT' + W_{sea} L_f \quad \text{on } \Gamma_W, \quad (10.39)$$

which may be evaluated using Equation (10.11) if $T_0 = 0$.

Basal regions containing overhead temperature at the temperature melting point T_m are defined by the coefficient

$$\alpha_w = \begin{cases} 0, & \nabla T \cdot \mathbf{n} \neq \nabla T_m \cdot \mathbf{n} \quad \text{on } \Gamma_G, \\ 1, & \nabla T \cdot \mathbf{n} = \nabla T_m \cdot \mathbf{n} \quad \text{on } \Gamma_G, \end{cases} \quad (10.40)$$

or, using the continuity of internal water content W , by the coefficient

$$\alpha = \begin{cases} 0, & W = 0 \quad \text{on } \Gamma_G, \\ 1, & W > 0 \quad \text{on } \Gamma_G. \end{cases} \quad (10.41)$$

Coefficient (10.41) is stronger than (10.40), as it does not require calculation of derivatives and is thus not affected by low-resolution approximation errors arising from the discretization.

Coefficient (10.41) used in conjunction with enthalpy-gradient flux (10.18) and basal melt rate (10.36) combines sensible energy flux (10.29) and latent energy flux (10.34) over the entire grounded surface Γ_G :

$$\left(\frac{\kappa}{c} \nabla \theta \right) \cdot \mathbf{n} = \begin{cases} g_N, & \alpha = 0 \\ g_N - (k \nabla T_m) \cdot \mathbf{n} - \rho_w L_f F_b, & \alpha = 1 \end{cases}, \quad (10.42)$$

producing finally the basal energy flux

$$\left(\frac{\kappa}{c} \nabla \theta \right) \cdot \mathbf{n} = g_N - \alpha g_W \quad \text{on } \Gamma_G, \quad (10.43)$$

where $g_W = (k \nabla T_m) \cdot \mathbf{n} + \rho_w L_f F_b$. By stating basal energy flux boundary (10.43) in this manner, a continuous range of basal energy flux values across the entire grounded basal surface is allowed.

In areas with overlying temperate ice, it has been previously assumed (Aschwanden et al., 2012; Kleiner et al., 2015) that the flux of water – and therefore energy – out of the ice is in balance with the water gradient caused by basal melt, corresponding with the condition $F_b = M_b \rho / \rho_w$ in energy-flux (10.43) and

$$\left(\frac{\kappa}{c} \nabla \theta \right) \cdot \mathbf{n} = g_N - \alpha g_N \quad \text{on } \Gamma_G. \quad (10.44)$$

By the strict use of this boundary condition, strain-heating may increase the water content of interior ice to abnormally high levels (Aschwanden et al., 2012). In contrast, by using generalized basal energy flux (10.43) while allowing the possibility for $F_b > M_b \rho / \rho_w$, the energy flux across the basal surface is able to adapt to large quantities of internally-generated water produced by internal friction (8.11).

Therefore, provided that the non-advective water-flux coefficient v in water flux (10.34) is greater than zero – corresponding to an enthalpy-gradient coefficient (10.18) with $k_0 < \infty$ in temperate regions – a procedure for choosing an appropriate value of F_b in (10.43) defines a mechanism for enforcing maximum water retention demand (10.10).

10.4 Exploring basal-melting-rate

As discussed in §10.3, basal melting rate (10.36) is only valid in regions where $\theta \geq \theta_m$ and is positive only when

$$q_{geo} + q_{fric} > (k_i \nabla T_m) \cdot \mathbf{n}. \quad (10.45)$$

Digging deeper into the pressure-melting gradient,

$$\begin{aligned} (k_i \nabla T_m) \cdot \mathbf{n} &= (k_i \gamma \nabla p) \cdot \mathbf{n} \\ &= k_i \gamma \nabla(\delta p) \cdot \mathbf{n} + k_i \gamma (\nabla(\rho g(S - z))) \cdot \mathbf{n} \\ &= k_i \gamma \nabla(\delta p) \cdot \mathbf{n} + k_i \gamma \rho g \left(\frac{\partial S}{\partial x} n_x + \frac{\partial S}{\partial y} n_y + n_z \right), \end{aligned}$$

where Clausius-Clapeyron relationship (10.12) has been applied and basal pressure p was separated into cryostatic $\rho g H$ with z -varying thickness $H = S - z$ for surface height S , and dynamic δp terms. Next, applying the low basal-slope requirement of first-order momentum Blatter (1995); Pattyn (2003), $\mathbf{n} \approx [0 \ 0 \ -1]^\top$ and

$$(k_i \nabla T_m) \cdot \mathbf{n} \approx -k_i \gamma \frac{\partial \delta p}{\partial z} - k_i \gamma \rho g = -k_i \gamma \left(\frac{\partial \delta p}{\partial z} + \rho g \right). \quad (10.46)$$

Therefore, because q_{geo} and q_{fric} (10.25) are both positive-definite functions, refreeze only occurs in regions where the dynamic pressure gradient is able to overcome the geothermal and frictional energy flux adjusted by a small scalar value,

$$q_{geo} + q_{fric} > k_i \gamma \left(\frac{\partial \delta p}{\partial z} + \rho g \right) \implies \text{refreezing}. \quad (10.47)$$

Note also that in the case of cryostatic assumptions, δp is zero and refreeze condition (10.47) simplifies to $q_{geo} + q_{fric} > k_i \gamma \rho g$. Hence with an average geothermal flux of $\mathcal{O}(10^{-2})$ W m⁻², $q_{fric} \gg q_{geo}$, and $k_i \gamma \rho g = \mathcal{O}(10^{-3})$ W m⁻², refreeze is unlikely to occur under these assumptions. Therefore, the full-Stokes momentum balance must be applied in order to properly identify refreezing or melt.

10.5 Weak energy approximation

The variational and weak form of the steady-state ($\partial_t \theta = 0$) version of energy balance (10.20) and associated boundary conditions (10.38, 10.39, 10.43) is constructed by taking the inner product of the residual

$$\mathcal{R}(\theta, F_b) = \rho \mathbf{u} \cdot \nabla \theta - \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \theta - \frac{\kappa}{c} \nabla \cdot \nabla \theta - Q. \quad (10.48)$$

with the test function $\psi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ (see test space (1.11)), integrating over the entire ice-sheet volume Ω ,

$$\begin{aligned} - \int_{\Omega} Q \psi \, d\Omega + \int_{\Omega} \rho \mathbf{u} \cdot \nabla \theta \psi \, d\Omega - \int_{\Omega} \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \theta \psi \, d\Omega \\ + \int_{\Omega} \left(\frac{\kappa}{c} \right) \nabla \theta \cdot \nabla \psi \, d\Omega - \int_{\Gamma} \left(\frac{\kappa}{c} \nabla \theta \right) \cdot \mathbf{n} \psi \, d\Gamma = 0, \end{aligned} \quad (10.49)$$

where the diffusive term has been integrated by parts. Because outward flux terms over essential boundaries vanish (see test space (1.11)), the boundary integral over Γ is reduced to an integral over Γ_G using energy flux (10.43),

$$\int_{\Gamma} \left(\frac{\kappa}{c} \nabla \theta \right) \cdot \mathbf{n} \psi \, d\Gamma = \int_{\Gamma_G} (g_N - \alpha g_W) \psi \, d\Gamma_G. \quad (10.50)$$

10.5.1 Numerical stabilization

Numerical instabilities will manifest in areas where the transport of energy is dominated by advection (see §5.5). Hence stabilization is required to reduce non-physical oscillations resulting from solving Galerkin-form (10.49).

To begin, the linear differential operator associated with problem (10.20) is

$$\mathcal{L}\theta = \rho \mathbf{u} \cdot \nabla \theta - \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \theta - \frac{\kappa}{c} \nabla \cdot \nabla \theta. \quad (10.51)$$

On close inspection, the advective part of this operator is

$$\mathcal{L}_{adv}\theta = \tilde{\mathbf{u}} \cdot \nabla \theta, \quad (10.52)$$

with quasi-velocity

$$\tilde{\mathbf{u}} = \rho \mathbf{u} - \nabla \left(\frac{\kappa}{c} \right). \quad (10.53)$$

The conductive gradient term $\nabla(\kappa/c)$ of the energy-flux therefore contributes in an advective sense to the transport of energy; energy transport increases as the magnitude of the conductive gradient increases. For example, if the ice is taken – without loss of generality – as stationary, $\mathbf{u} = \mathbf{0}$ and energy transport occurs by diffusion and quasi-advection in the down-gradient direction of the conduction term κ/c .

Next, the stabilized form of internal-energy balance (10.49) with linear operator (10.51) using general stabilized form (5.16) with test function ψ and intrinsic-time parameter τ_{IE} is

$$(\psi, \mathcal{L}\theta) + (\mathbb{L}\psi, \tau_{IE}(\mathcal{L}\theta - Q)) = (\psi, Q), \quad (10.54)$$

where operator \mathbb{L} is a differential operator chosen from

$$\mathbb{L} = +\mathcal{L} \quad \text{Galerkin/least-squares (GLS)} \quad (10.55)$$

$$\mathbb{L} = +\mathcal{L}_{adv} \quad \text{SUPG} \quad (10.56)$$

$$\mathbb{L} = -\mathcal{L}^* \quad \text{subgrid-scale model (SSM)} \quad (10.57)$$

with adjoint of operator \mathcal{L} denoted \mathcal{L}^* .

Making the appropriate substitutions in SUPG parameter (5.20) results in the intrinsic-time parameter

$$\tau_{IE} = \frac{h \xi(P_\epsilon)}{2 \|\tilde{\mathbf{u}}\|}, \quad P_\epsilon = \frac{h \|\tilde{\mathbf{u}}\|}{2 \rho \Xi}, \quad (10.58)$$

where h is the cell diameter, P_ϵ is the ratio of advective to diffusive flux coefficients, referred to as the *element Péclet* number, and $\xi(P_\epsilon)$ is a function that is dependent on the element-shape-functions utilized. For example, if the shape functions ψ are taken as the linear Lagrange elements described in §1.2.2, the accuracy-optimal function choice for ξ is given by (Brooks and Hughes, 1982)

$$\xi(P_\epsilon) = \coth(P_\epsilon) - \frac{1}{P_\epsilon}. \quad (10.59)$$

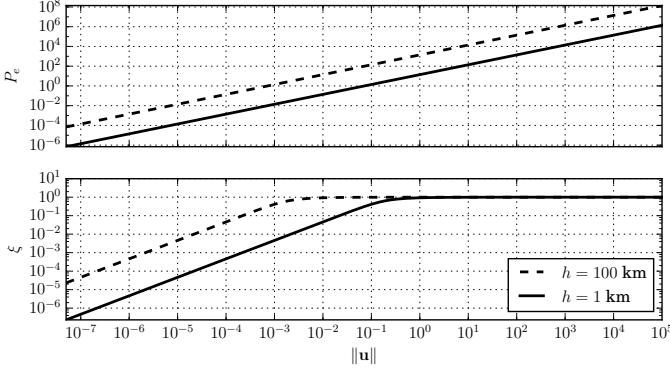


Figure 10.3: The element Péclet number P_ϵ (top) with $h = 1$ km (solid) and $h = 100$ km (dashed) over a range of velocity values with magnitude $\|\mathbf{u}\|$ in m a^{-1} appropriate to ice-sheets. The corresponding intrinsic time ξ multiplicative term to the SUPG formulation for linear-Lagrange elements (10.59) (bottom) becomes very close to unity after the ice speed gets above $\|\mathbf{u}\| \approx 2 \text{ cm a}^{-1}$ with $h = 1$ km.

It was later determined by Codina, Oñate, and Cervera (1992) that when quadratic shape functions are used, the accuracy-optimal function for ξ changes to

$$\xi_1(P_\epsilon) = \frac{1}{2} \left(\coth\left(\frac{P_\epsilon}{2}\right) - \frac{2}{P_\epsilon} \right)$$

$$\xi(P_\epsilon) = \frac{(3 + 3P_\epsilon \xi_1) \tanh(P_\epsilon) - (3P_\epsilon + P_\epsilon^2 \xi_1)}{(2 - 3\xi_1 \tanh(P_\epsilon)) P_\epsilon^2}. \quad (10.60)$$

Note that for any substantial ice flow, P_ϵ will be very large and thus $\xi \approx 1$ (Figure 10.3). Additionally, note that if linear-Lagrange elements are used, the application of SUPG stabilization (10.56) and GLS stabilization (10.55) in form (10.54) are identical.

Therefore, using boundary integral (10.50) with variational form (10.49) in stabilized form (10.54), the problem consists of finding $\theta \in \mathcal{H}^1(\Omega)$ such that

$$\begin{aligned} & - \int_{\Omega} Q \psi \, d\Omega + \int_{\Omega} \rho \mathbf{u} \cdot \nabla \theta \psi \, d\Omega - \int_{\Omega} \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \theta \psi \, d\Omega \\ & + \int_{\Omega} \left(\frac{\kappa}{c} \right) \nabla \theta \cdot \nabla \psi \, d\Omega - \int_{\Gamma_G} (g_N - \alpha g_W) \psi \, d\Gamma_G \\ & + \int_{\Omega} \tau_{\text{IE}}(\mathbb{L}\psi) (\mathcal{L}\theta - Q) \, d\Omega - \int_{\Omega} Q \psi \, d\Omega = 0, \end{aligned} \quad (10.61)$$

for all $\psi \in \mathcal{H}^1(\Omega)$ subject to the remaining essential boundary conditions (10.38) and (10.39).

10.5.2 Energy balance discretization

For a mesh with N_n vertices and N_Γ essential exterior vertices corresponding with Dirichlet boundaries (10.38, 10.39),

the approximation

$$\theta \approx \theta_h = \sum_{j=1}^{N_n} \theta_j \psi_j + \sum_{j=N_n+1}^{N_n+N_\Gamma} \theta_j \psi_j \quad (10.62)$$

defines an expansion of θ with unknown coefficients θ_j associated with the trial functions ψ_j (see trial space (1.10)). Inserting approximation (10.62) into (10.61) results in the matrix-vector set of equations

$$\mathbf{r} = \mathcal{C}\theta - \mathcal{K}\theta + \mathcal{D}\theta + \mathcal{S}\theta + \mathbf{f}^{\text{ext}} - \mathbf{f}^{\text{int}} - \mathbf{f}^{\text{stz}}, \quad (10.63)$$

where for each test function ψ_i with $i, j = 1, 2, \dots, N_n$,

$$\mathcal{C}_{ij} = \int_{\Omega} \rho \mathbf{u} \cdot \nabla \psi_j \psi_i \, d\Omega \quad \leftarrow \text{advection} \quad (10.64)$$

$$\mathcal{K}_{ij} = \int_{\Omega} \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \psi_j \psi_i \, d\Omega \quad \leftarrow \text{cond. grad.} \quad (10.65)$$

$$\mathcal{D}_{ij} = \int_{\Omega} \left(\frac{\kappa}{c} \right) \nabla \psi_j \cdot \nabla \psi_i \, d\Omega \quad \leftarrow \text{diffusion} \quad (10.66)$$

$$\mathcal{S}_{ij} = \int_{\Omega} \tau_{\text{IE}}(\mathbb{L}\psi_i) (\mathcal{L}\psi_j) \, d\Omega \quad \leftarrow \text{stab'z'tion} \quad (10.67)$$

$$\mathbf{f}_i^{\text{ext}} = \int_{\Gamma_G} (g_N - \alpha g_W) \psi_i \, d\Gamma_G \quad \leftarrow \text{energy flux} \quad (10.68)$$

$$\mathbf{f}_i^{\text{int}} = \int_{\Omega} Q \psi_i \, d\Omega \quad \leftarrow \text{strain heat} \quad (10.69)$$

$$\mathbf{f}_i^{\text{stz}} = \int_{\Omega} \tau_{\text{IE}}(\mathbb{L}\psi_i) Q \, d\Omega \quad \leftarrow \text{stab'z'tion}, \quad (10.70)$$

and \mathbf{r} is the residual error vector, defining a *weak solution* θ to variational form (10.61) when $\|\mathbf{r}\| = 0$. When using approximation (10.62), the sum involving the N_Γ exterior vertices will produce a set of matrices with identical properties as (10.64 – 10.66), but where the unknown coefficients θ_j , $j = N_n + 1, N_n + 2, \dots, N_n + N_\Gamma$ are known from the surface essential boundaries (10.38 – 10.39). Thus, the last sum in approximation (10.62) interpolates the boundary data onto the finite-element basis (Elman, Silvester, and Wathen, 2005).

If an identical discrete basis is used for both ψ_i and ψ_j , θ_h defined by (10.62) corresponds with *Bubnov-Galerkin* approximation. In such a case, it is easily seen that \mathcal{D} is symmetric, and as it turns out, positive-definite (Elman, Silvester, and Wathen, 2005). The same cannot be said of \mathcal{C} and \mathcal{K} . However, symmetry is added back to linear system (10.63) by the term \mathcal{S} . For example, if linear Lagrange elements are used as a basis for ψ and streamline-upwind/Petrov-Galerkin stabilization operator (10.56) is used,

$$\begin{aligned} \mathcal{S}_{ij} &= \int_{\Omega} \tau_{\text{IE}}(\mathcal{L}_{\text{adv}}\psi_i) (\mathcal{L}\psi_j) \, d\Omega \\ &= \int_{\Omega} \tau_{\text{IE}} \left(\left(\rho \mathbf{u} + \nabla \left(\frac{\kappa}{c} \right) \right) \cdot \nabla \psi_i \right) \left(\left(\rho \mathbf{u} + \nabla \left(\frac{\kappa}{c} \right) \right) \cdot \nabla \psi_j \right) \, d\Omega \\ &= \int_{\Omega} \tau_{\text{IE}}(\tilde{\mathbf{u}} \cdot \nabla \psi_i) (\tilde{\mathbf{u}} \cdot \nabla \psi_j) \, d\Omega, \end{aligned} \quad (10.71)$$

where quasi-velocity (10.53) has been applied and the fact that second-derivatives of linear element shape-functions are zero. Matrix (10.71) is symmetric-positive-definite, and thus the addition of this term in (10.63) has the effect of increasing the stability or *coercivity* of the linear system.

An algorithm well suited for the solution of problems possessing non-symmetric matrices such as these is the *generalized minimum residual method* (GMRES). This procedure is one of many *Krylov subspace methods*, which iteratively reduces the energy norm of the error. However, because the non-advective flux coefficient κ is discontinuous, depending on the unknown θ , and also because thermal properties (10.8) and (10.9) are non-linear with respect to θ , system (10.63) is non-linear. Thus, a linearization of this system is required. One such linearization is *Newton's method*, which iteratively reduces residual (10.63) by solving for the direction of decent with respect to the solution space of θ (investigate §6.1).

The source code of CSLVR uses an implementation similar to Code Listing 10.1.

Code Listing 10.1: CSLVR source code contained in the Enthalpy class.

```
# Define test and trial functions :
psi = TestFunction(model.Q)
dtheta = TrialFunction(model.Q)
theta = Function(model.Q, name='energy.theta')

# momentum-dependent properties :
U = momentum.velocity()
epsdot = momentum.effective_strain_rate(U) + model.eps_reg
eta = momentum.viscosity(U)

# internal friction (strain heat) :
Q_s = 4 * eta * epsdot

# coefficient for non-advective water flux (enthalpy-gradient) :
k_c = conditional( gt(W, 0.0), model.k_0, 1 )

# thermal conductivity and heat capacity (Greve and Blatter 2009) :
ki = 9.828 * exp(-0.0057*T)
ci = 146.3 + 7.253*T

# bulk properties :
k = (1 - W)*ki + W*kw # bulk thermal conductivity
c = (1 - W)*ci + W*cw # bulk heat capacity
rho = (1 - W)*rhoi + W*rhow # bulk density
kappa = spy * k_c * k # discontinuous with water, J/(a*m*K)
Xi = kappa / (rho*c) # bulk enthalpy-gradient diffusivity

# basal heat-flux natural boundary condition :
q_fric = beta + inner(U,U)
g_w = model.gradTm_B + rhow*L*Fb
g_n = q_geo + q_fric
g_b = g_n - alpha*g_w

# the Peclet number :
Ut = rho*U - grad(kappa/c)
Unorm = sqrt(dot(Ut, Ut) + DOLFIN_EPS)
PE = Unorm*h/(2*kappa/c)

# for linear elements :
if model.order == 1:
    xi = 1/tanh(PE) - 1/PE

# for quadratic elements :
if model.order == 2:
    xi_1 = 0.5*(1/tanh(PE) - 2/PE)
    xi = ((3 + 3*PE*xi_1)*tanh(PE) - (3*PE + PE**2*xi_1)) \
        / ((2 - 3*xi_1*tanh(PE))*PE**2)

# intrinsic time parameter :
tau = h*xi / (2 + Unorm)

# the linear differential operator for this problem :
def Lu(u):
    Lu = + rho * dot(U, grad(u)) \
        - kappa/c * div(grad(u)) \
        - dot(grad(kappa/c), grad(u))
    return Lu

# the advective part of the operator :
def L_adv(u):
    return dot(Ut, grad(u))

# the adjoint of the operator :
def L_star(u):
    Ls = - dot(U, grad(u)) \
        - Xi * div(grad(u)) \
        + 1/rho * dot(grad(kappa/c), grad(u))
    return Ls

# use streamline-upwind/Petrov-Galerkin stabilization :
if stabilization_method == 'SUPG':
    s = " - using streamline-upwind/Petrov-Galerkin stabilization - "
    LL = lambda x: L_adv(x)
# use Galerkin/least-squares stabilization :
elif stabilization_method == 'GLS':
    s = " - using Galerkin/least-squares stabilization - "
    LL = lambda x: Lu(x)
# use subgrid-scale-model stabilization :
elif stabilization_method == 'SSM':
    LL = lambda x: L_star(x)
print_text(s, cls=self)

self.theta_a = + rho * dot(U, grad(dtheta)) * psi * dx \
    + kappa/c * inner(grad(psi), grad(dtheta)) * dx \
    - dot(grad(kappa/c), grad(dtheta)) * psi * dx \
    + inner(LL(psi), tau * Q_s) * dx

self.theta_L = + g_b * psi * dBed_g \
    + Q_s * psi * dx \
    + inner(LL(psi), tau * Q_s) * dx

# surface boundary condition :
self.theta_bc = []
self.theta_bc.append( DirichletBC(Q, theta_surface,
```

```
model.ff, model.GAMMA_S_GND)
self.theta_bc.append( DirichletBC(Q, theta_surface,
model.ff, model.GAMMA_S_FLT) )
self.theta_bc.append( DirichletBC(Q, theta_surface,
model.ff, model.GAMMA_U_GND) )
self.theta_bc.append( DirichletBC(Q, theta_surface,
model.ff, model.GAMMA_U_FLT) )

# apply T_melt conditions of portion of ice in contact with water :
self.theta_bc.append( DirichletBC(Q, theta_float,
model.ff, model.GAMMA_B_FLT) )
self.theta_bc.append( DirichletBC(Q, theta_float,
model.ff, model.GAMMA_L_UDR) )

# form the solver parameters :
self.solve_params = self.default_solve_params()

def default_solve_params(self):
    """
    Returns a set of default solver parameters that yield good performance
    """
    params = {'solver' : ('linear_solver', 'gmres',
        'preconditioner' : 'amg'),
        'use_surface_climate' : False}
    return params

def solve(self, annotate=False):
    """
    Solve the energy equations, saving enthalpy to model.theta, temperature
    to model.T, and water content to model.W.
    """
    solve(self.theta_a == self.theta_L, self.theta, self.theta_bc,
        solver_parameters = self.solve_params['solver'], annotate=annotate)
```

10.6 Water content optimization

In the absence of a constitutive relation for basal water discharge F_b , system of equations (10.20, 10.38, 10.39, 10.43) is ill-posed. However, this problem can be overcome using methods from control theory (Bryson and Ho, 1975; MacAyeal, 1993; Nocedal and Wright, 2000). Notice that it is expected that for very low basal water content W the discharge of water from the base of the ice-sheet be small; likewise, in areas of abnormally high basal water content it is expected that the discharge of water from the ice be high. Thus it is desired to minimize the difference between water content W and maximum water content W_c as given by demand (10.10) over the entire basal surface. Mathematically, this can be stated in terms of the *state* parameter θ as minimizing the L^2 objective functional

$$\mathcal{J}(\theta) = \frac{1}{2} \int_{\Gamma_G} (\theta - \theta_c)^2 d\Gamma_G, \quad (10.72)$$

where $\theta_c = \theta_m + W_c L_f$ is the maximum energy associated with maximum water content demand (10.10). This functional will be minimized in two ways: first, by minimizing the flow of water out of the ice in regions with $\theta < \theta_c$, corresponding with the lower bound $F_b = 0$ in basal energy flux (10.43); and second, by maximizing the flow of water out of the ice at regions with $\theta > \theta_c$, corresponding with $F_b > M_b \rho / \rho_w$. The role of F_b in basal energy flux (10.43) is hence the *control* parameter for the minimization of objective (10.72).

For additional illustration, recall that the inward-directed flow of energy-flux (10.43) is maximal for lower bound $F_b = 0$. Therefore, at regions with $T < T_m$, it is expected that $F_b = 0$. Furthermore, because parameter α defined by (10.41) eliminates any basal water discharge over cold regions in energy flux (10.43), this expectation is automatically satisfied and integration across the entire grounded basal domain Γ_G by objective (10.72) is justified. Notice also that an F_b which produces a minimum of objective (10.72) will of course affect the distribution of water content W in the mixture interior, and thus also affect the enhancement of flow as evident by the water-content dependence of rate-factor (10.22). Because of

this, it is either necessary to re-compute the momentum balance for each optimization of θ , or combine both energy and momentum into a single mixed formulation.

The minimization of objective (10.72), solution of variational problem (10.61, 10.38, 10.39), and satisfaction of the positivity constraint of basal water flux $F_b \geq 0$ can be stated as a constrained optimization problem analogous to that presented in Chapter 7. Thus we state the problem in the form

$$\min_{\theta} \mathcal{J}(\theta) \quad \text{subject to} \quad \begin{cases} \mathcal{R}(\theta, F_b) = 0, \\ F_b \geq 0, \end{cases} \quad (10.73)$$

where \mathcal{R} is the residual, or in this context *forward model* defined by (10.48). The energy Lagrangian associated with problem (10.73) is (see Chapter 7)

$$\mathcal{L}(\theta, F_b, \lambda) = \mathcal{J}(\theta, F_b) + (\lambda, \mathcal{R}(\theta, F_b)), \quad (10.74)$$

where the notation $(f, g) = \int_{\Omega} f g d\Omega$ is the inner product. Using Lagrangian (10.74), the first necessary condition in (7.2) is satisfied when *adjoint variable* λ is chosen – say $\lambda = \lambda^*$ – such that for a given energy state θ and control parameter F_b ,

$$\lambda^* = \operatorname{argmin}_{\lambda} \left\| \frac{\delta}{\delta \theta} \mathcal{L}(\theta, F_b; \lambda) \right\|. \quad (10.75)$$

This λ^* may then be used in condition (7.7) to calculate the direction of decent of \mathcal{L} with respect to the control variable F_b for a given energy state θ and adjoint variable λ^* ,

$$G = \frac{\delta}{\delta F_b} \mathcal{L}(\theta, F_b, \lambda^*). \quad (10.76)$$

This *Gâteaux derivative* provides a direction which basal water discharge F_b may follow in order to satisfy the second condition in (7.2) and thus minimize objective functional (10.72).

10.6.1 Variations

Lagrangian (10.74) is formed by taking the inner product of the adjoint variable λ with forward model (10.48), integrating over the entire domain Ω , integrating the diffusive term by parts, and adding stabilization,

$$\begin{aligned} \mathcal{L}(\theta, F_b, \lambda) = & + \frac{1}{2} \int_{\Gamma_G} (\theta - \theta_c)^2 d\Gamma_G \\ & + \int_{\Omega} \rho \mathbf{u} \cdot \nabla \theta \lambda d\Omega - \int_{\Omega} Q \lambda d\Omega \\ & - \int_{\Omega} \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \theta \lambda d\Omega + \int_{\Omega} \left(\frac{\kappa}{c} \right) \nabla \theta \cdot \nabla \lambda d\Omega \\ & - \int_{\Gamma_G} (g_N - \alpha g_W) \lambda d\Gamma_G \\ & + \int_{\Omega} \tau_{IE}(\mathbb{L}\lambda) (\mathcal{L}\theta - Q) d\Omega. \end{aligned} \quad (10.77)$$

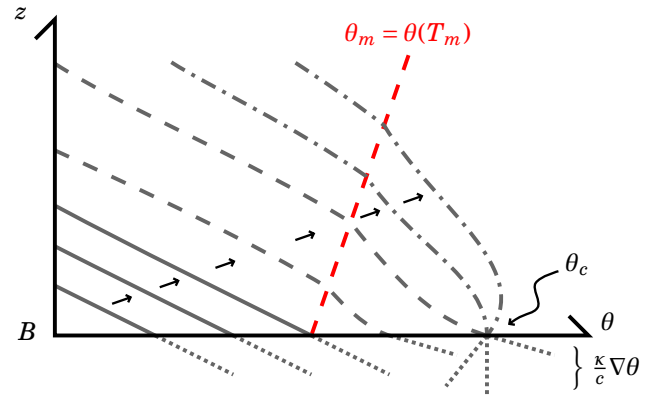


Figure 10.4: Illustration of the transition from cold ice to temperate. The arrows point in the direction of increasing energy profiles, with cold ice profiles (solid gray), temperate ice with basal water contents less than W_c (dashed gray), and temperate ice which would have basal water contents higher than W_c without some amount of basal water discharge F_b (dashed-dotted gray). Note that the gradient in θ increases once the ice becomes temperate, as required when $(k \nabla T_m) \cdot \mathbf{n} < 0$ in basal energy flux (10.43).

Therefore, the first variation of \mathcal{L} with respect to θ in the direction ψ is

$$\begin{aligned} \frac{\delta \mathcal{L}}{\delta \theta} = & + \int_{\Gamma_G} (\theta - \theta_c) \psi d\Gamma_G + \int_{\Omega} \rho \mathbf{u} \cdot \nabla \psi \lambda d\Omega \\ & - \int_{\Omega} \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \psi \lambda d\Omega + \int_{\Omega} \left(\frac{\kappa}{c} \right) \nabla \psi \cdot \nabla \lambda d\Omega \\ & + \int_{\Omega} \tau_{IE}(\mathbb{L}\lambda) \left(\rho \mathbf{u} \cdot \nabla \psi - \nabla \left(\frac{\kappa}{c} \right) \cdot \nabla \psi - \left(\frac{\kappa}{c} \right) \nabla \cdot \nabla \psi \right) \lambda d\Omega. \end{aligned} \quad (10.78)$$

while the first variation of \mathcal{L} with respect to F_b is

$$G(F_b, \lambda) = \frac{\delta \mathcal{L}}{\delta F_b} = \int_{\Gamma_G} \psi \alpha \rho_w L_f \lambda d\Gamma_G. \quad (10.79)$$

10.6.2 Energy optimization procedure

To determine an optimal value of basal water discharge F_b , a variation of a primal-dual-interior-point algorithm with a filter-line-search method implemented by the IPOPT framework (Wächter and Biegler, 2006) may be used (read §7.2). In the context of problem (10.73), the algorithm implemented by IPOPT computes approximate solutions to a sequence of barrier problems

$$\min_{F_b} \left\{ \varphi_{\mu}(\theta, F_b) = \mathcal{J}(\theta) - \mu \sum_{i=1}^{N_n} \ln(F_b^i) \right\} \quad (10.80)$$

for a decreasing sequence of barrier parameters μ converging to zero, and N_n is the number of degrees of freedom of the finite-element mesh. For further details of this algorithm, see §7.2.

We defer an example of the energy-optimization and solution process until Chapter 11 which describes the thermo-mechanical coupling of energy θ and momentum (\mathbf{u}, p) . The CSLVR implementation is shown in Code Listing 10.2.

10.7 Effect of discontinuous energy conductivity

A cause of concern for polythermal glaciologists is to correctly calculate the position of the CTS. While this analysis addresses a separate issue pertaining to the energy balance – the basal boundary condition – it is important to note that this method is compatible with alternative solution methods of the energy balance equations.

First, it is expected that for a lower non-advective water diffusion coefficient ν , the basal water discharge F_b must adapt to reduce the water generated from strain-heating, located interior to the ice. Of course, if $\nu = 0$ as in Greve and Blatter (2009), no amount increase in F_b will reduce the internal water content due to the fact that basal-latent-heat-flux-boundary-condition (10.34) is zero. Likewise, for large values of ν , water will be very efficiently routed and W will be very sensitive to perturbations in F_b .

For any given energy-balance formulation using $\nu > 0$ implementing the procedure of §10.6, as the outward flux of water increases, intra-ice water is moved from the interior to the bed until the gradient in θ reaches the point that cost functional (10.72) cannot be decreased further.

Code Listing 10.2: CSLVR source code contained in the Energy class for solving the water-content-optimization procedure of §10.6.

```
def optimize_water_flux(self, max_iter, bounds, method='ipopt',
                        adj_save_vars=None, adj_callback=None):
    """
    determine the correct basal-water flux.
    """
    s = ' ::: optimizing for water-flux in %i maximum iterations ::: '
    print_text(s % max_iter, cls=self)

    model = self.model

    # reset entire dolfin-adjoint state :
    adj_reset()

    # starting time :
    t0 = time()

    # need this for the derivative callback :
    global counter
    counter = 0

    # functional lists to be populated :
    global Rs, Js, Ds
    Rs = []
    Js = []
    Ds = []

    # now solve the control optimization problem :
    s = " ::: starting adjoint-control optimization with method '%s' ::: "
    print_text(s % method, cls=self)

    def eval_cb(I, Fb):
        s = ' ::: adjoint objective eval post callback function ::: '
        print_text(s, cls=self)
        print_min_max(I, 'I', cls=self)
        print_min_max(Fb, 'Fb', cls=self)

    # objective gradient callback function :
    def deriv_cb(I, dI, Fb):
        global counter, Rs, Js
        if method == 'ipopt':
            s0 = '>>>'
            s1 = 'iteration %i (max %i) complete'
            s2 = '<<<'
            text0 = get_text(s0, 'red', 1)
            text1 = get_text(s1 % (counter, max_iter), 'red')
            text2 = get_text(s2, 'red', 1)
            if MPI.rank(mpi_comm_world())==0:
                print text0 + text1 + text2
                counter += 1
            s = ' ::: adjoint obj. gradient post callback function ::: '
            print_text(s, cls=self)
            print_min_max(dI, 'dI/Fb', cls=self)

        # update the DA current velocity to the model for evaluation
        # purposes only; the model.assign_variable function is
        # annotated for purposes of linking physics models to the adjoint
```

```
# process :
theta_opt = DolfinAdjointVariable(model.theta).tape_value()
model.init_theta(theta_opt, cls=self)

# print functional values :
model.Fb.assign(Fb, annotate=False)
ftnls = self.calc_functionals()
D = self.calc_misfit()

# functional lists to be populated :
Rs.append(ftnls[0])
Js.append(ftnls[1])
Ds.append(D)

# call that callback, if you want :
if adj_callback is not None:
    adj_callback(I, dI, Fb)

# solve the momentum equations with annotation enabled :
s = ' ::: solving forward problem for dolfin-adjoint annotation ::: '
print_text(s, cls=self)
self.solve(annotate=True)

# get the cost, regularization, and objective functionals :
I = self.J
try:
    I += self.R
except AttributeError:
    print_text(' - not using regularization -', cls=self)

# define the control variable :
m = Control(model.Fb, value=model.Fb)

# state the minimization problem :
F = ReducedFunctional(Functional(I), m, eval_cb_post=eval_cb,
                      derivative_cb_post=deriv_cb)

# optimize with scipy's fmin_l_bfgs_b :
if method == 'l_bfgs_b':
    out = minimize(F, method="L-BFGS-B", tol=1e-9, bounds=bounds,
                  options={"disp": True,
                           "maxiter": max_iter,
                           "gtol": 1e-5})
    Fb_opt = out[0]

# or optimize with IPOPT (preferred) :
elif method == 'ipopt':
    try:
        import pyipopt
    except ImportError:
        info_red("You do not have IPOPT and/or pyipopt installed.
                  When compiling IPOPT, make sure to link against HSL,
                  as it is a necessity for practical problems.")

    raise

    problem = MinimizationProblem(F, bounds=bounds)
    parameters = {"tol": 1e-8,
                  "acceptable_tol": 1e-6,
                  "maximum_iterations": max_iter,
                  "print_level": 5,
                  "ma97_order": "metis",
                  "ma86_order": "metis",
                  "linear_solver": "ma57"}

    solver = IPOPTSolver(problem, parameters=parameters)
    Fb_opt = solver.solve()

# let's see it :
print_min_max(Fb_opt, 'Fb_opt')

# extrude the flux up and make the optimal control variable available :
Fb_ext = model.vert_extrude(Fb_opt, d='up')
model.init_Fb(Fb_ext, cls=self)
#Control(model.Fb).update(Fb_ext) # FIXME: does this work?

# save state to unique hdf5 file :
if isinstance(adj_save_vars, list):
    s = ' ::: saving variables in list arg adj_save_vars ::: '
    print_text(s, cls=self)
    out_file = model.out_dir + 'w_opt.h5'
    foutput = HDF5File(mpi_comm_world(), out_file, 'w')
    for var in adj_save_vars:
        model.save_hdf5(var, f=foutput)
    foutput.close()

# calculate total time to compute
tf = time()
s = tf - t0
m = s / 60.0
h = s / 60.0
s = s % 60
m = m % 60
text = "time to optimize for water flux: %02d:%02d:%02d" % (h,m,s)
print_text(text, 'red', 1)

# save all the objective functional values :
d = model.out_dir + 'objective_ftnls_history/'
s = ' ::: saving objective functionals to %s ::: '
print_text(s % d, cls=self)
if model.MPI_rank==0:
    if not os.path.exists(d):
        os.makedirs(d)
    np.savetxt(d + 'time.txt', np.array([tf - t0]))
    np.savetxt(d + 'Rs.txt', np.array(Rs))
    np.savetxt(d + 'Js.txt', np.array(Js))
    np.savetxt(d + 'Ds.txt', np.array(Ds))

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_yscale('log')
ax.set_ylabel(r'$\mathscr{J}$\left(\theta\right)$')
ax.set_xlabel(r'iteration')
ax.plot(np.array(Js), 'r-', lw=2.0)
plt.grid()
plt.savefig(d + 'J.png', dpi=100)
plt.close(fig)

try:
    R = self.R
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_yscale('log')
    ax.set_ylabel(r'$\mathscr{R}$\left(\alpha\right)$')
    ax.set_xlabel(r'iteration')
    ax.plot(np.array(Rs), 'r-', lw=2.0)
    plt.grid()
    plt.savefig(d + 'R.png', dpi=100)
    plt.close(fig)
except AttributeError:
    pass

fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_yscale('log')
ax.set_ylabel(r'$\mathscr{D}$\left(\theta\right)$')
ax.set_xlabel(r'iteration')
ax.plot(np.array(Ds), 'r-', lw=2.0)
```

```
plt.grid()
plt.savefig(d + 'D.png', dpi=100)
plt.close(fig)
```


Chapter 11

Thermo-mechanical coupling

Coupling between the momentum-balance models described in Chapter 9 and the energy-balance model of Chapter 10 – referred to as *thermo-mechanical coupling* (TMC) – is accomplished here via fixed-point iteration, a process which generates approximations of velocity \mathbf{u} , pressure p , and energy θ that eventually converge to a stationary point. This stationary point is attained when the norm of the difference between two successive approximations of θ are below a specified tolerance.

First, as discussed in §9.6, because effective strain-rates (8.8), (9.21), (9.37), and (9.52) are non-linear in \mathbf{u} , momentum systems (9.13), (9.32), (9.48), and (9.63) are linearized using Newton’s method (see §6.1).

To begin, it has been observed that this linearization will converge consistently provided that the current velocity guess \mathbf{u} is sufficiently far from a stationary point, and so we initialize \mathbf{u} to zero at the start of every iteration. Once \mathbf{u} and p have been obtained by solving the momentum balance, the water-content optimization procedure described in the previous section is performed. This procedure results in an optimal distribution of water for a given friction β , and thus an improved estimate of rate factor (10.22) to be used in the subsequent iteration’s momentum formulation (Algorithm 4).

Returning to non-linear energy balance discretization (10.63), note that a solution process for this non-linear system – such as Newton’s method – requires several intermediate solutions of (10.63) be solved. Thus, the non-linearity with respect to θ present in thermal conductivity (10.8), heat capacity (10.9), energy-flux (10.18), and rate-factor (10.22) may be eliminated if instead the discontinuities are evaluated with regard to the previous TMC iteration’s pressure-melting point. This simplification saves considerable time, especially when considering the many forward-model (10.48) solutions required by energy barrier problem (10.80).

It remains to properly define the temperate zone, and thus correct values for temperate zone marker α defined by (10.41). To accomplish this, α is initially set to zero across the entire basal surface, hence assuming that the ice is cold throughout, and basal energy source (10.26) is universally applied. The energy distribution resulting from the energy balance is then evaluated, and α is assigned a value of one along any facet containing energies above pressure-melting energy (10.13).

While this method of boundary marking is approximate,

note that for temperate regions where the pressure-melting temperature decreases when moving up the ice column, flux of water (10.34) with zero basal water discharge F_b will be larger than basal energy source (10.26), and will thus clearly be temperate (Figure 10.4). It follows that this marking method has the potential to incorrectly mark boundaries only in areas where the pressure-melting point *increases* when moving up from the basal surface and with negligible water transported by advection to its location. The temperate basal boundary marking process and thermal-parameter linearization scheme is outlined by Algorithm 5.

Note in Algorithm 4 that we initialize basal-water discharge F_b to the value $M_b \rho / \rho_w$ – consistent with zero-energy-flux boundary condition (10.44) – prior to solving F_b -optimization problem (10.73). This has the effect of starting the optimization process at the same point for each iteration of TMC Algorithm 4, and leads to better convergence characteristics of the algorithm.

The implementation used by CSLVR for Algorithms 4 and 5 are shown in Code Listing 11.1 and 11.2, respectively.

Code Listing 11.1: CSLVR source code contained in the `Model` class used to perform the thermo-mechanical coupling between the `Momentum` and `Energy` classes.

```
def thermo_solve(self, momentum, energy, vop_kwargs,
                 callback=None, atol=1e2, rtol=1e0, max_iter=50,
                 iter_save_vars=None, post_tmc_save_vars=None,
                 starting_i=1):
    """
    Perform thermo-mechanical coupling between momentum and energy.

    Args:
        :momentum: a :class:`momentum.Momentum` instance
        :energy: a :class:`energy.Energy` instance. Currently this only
            works for :class:`energy.Enthalpy`
        :vop_kwargs: a :py:class:`dict` of arguments for water-optimization
            method :func:`energy.Energy.optimize_water_flux`
        :callback: a function that is called back at the end of each
            iteration
        :atol: absolute stopping tolerance
            :math:'a_{\{tol\}} \leq r = \|\text{Vert } \theta_{n-1} - \theta_n\| \text{Vert}'
        :rtol: relative stopping tolerance
            :math:'r_{\{tol\}} \leq \|\text{Vert } r_n - r_{n-1}\| \text{Vert}'
        :max_iter: maximum number of iterations to perform
        :iter_save_vars: python :py:class:`list` containing functions to
            save each iteration
        :starting_i: if you are restarting this process, you may start
            it at a later iteration.
    """
    s = '::: performing thermo-mechanical coupling with atol = %.2e, ' + \
        'rtol = %.2e, and max_iter = %i :::'
    print_text(s % (atol, rtol, max_iter), cls=self.this)

    from csivr import Momentum
    from csivr import Energy

    if not isinstance(momentum, Momentum):
        s = '>>> thermo_solve REQUIRES A 'Momentum' INSTANCE, NOT %s <<<'
        print_text(s % type(momentum), 'red', 1)
        sys.exit(1)

    if not isinstance(energy, Energy):
        s = '>>> thermo_solve REQUIRES AN 'Energy' INSTANCE, NOT %s <<<'
        print_text(s % type(energy), 'red', 1)
        sys.exit(1)

    # mark starting time :
    t0 = time()
```

Algorithm 4 – TMC fixed-point iteration

```

1: function TMC( $\beta, \theta, F_b$ )
2:    $a_{tol} := 100; n_{max} := 350; r := \infty; a := \infty; i := 1$ 
3:   while ( $a > a_{tol}$  or  $r > r_{tol}$ ) and  $i < n_{max}$  do
4:      $\mathbf{U} := [\mathbf{u} \ p]^\top \in (\mathcal{H}^1(\Omega))^4 = [0, p]^\top$ 
5:      $\mathbf{U}^* := \operatorname{argmin}_{\mathbf{U}} \left\| \delta_{\mathbf{U}} \mathcal{A}(\beta, \theta, F_b; \mathbf{U}) \right\|$ 
6:      $\text{TPU}(\mathbf{U}^*, \beta, F_b)$ 
7:      $q_{fric} := \beta \|\mathbf{u}\|^2$ 
8:      $M_b := \frac{q_{geo} + q_{fric} - (k \nabla T) \cdot \mathbf{n}}{L_f \rho}$ 
9:      $F_b := M_b \rho / \rho_w$ 
10:    if we want to optimize  $F_b$  then
11:       $F_b^* := \operatorname{argmin}_{F_b} \left\{ \varphi_\mu(F_b) \right\}$ 
12:    else
13:       $F_b^* := F_b$ 
14:    end if
15:     $\theta^* := \operatorname{argmin}_{\theta} \left\| \mathcal{R}(\mathbf{U}^*, \beta, F_b^*; \theta) \right\|$ 
16:     $a_n := \|\theta - \theta^*\|_2$ 
17:    if  $i = 1$  then
18:       $r := a_n$ 
19:    else
20:       $r := |a - a_n|$ 
21:    end if
22:     $\theta := \theta^*, F_b := F_b^*, i := i + 1; a := a_n$ 
23:  end while
24:  return  $\theta, F_b$ 
25: end function

```

Algorithm 5 – thermal parameters update (TPU)

```

1: function TPU( $\mathbf{U}, \beta, F_b$ )
2:    $a_{tol} := 100; n_{max} := 50; a := \infty; r := \infty; i := 1; \theta := 0$ 
3:    $\alpha := 0$ 
4:   while ( $a > a_{tol}$  or  $r > r_{tol}$ ) and  $i < n_{max}$  do
5:      $\theta^* := \operatorname{argmin}_{\theta} \left\| \mathcal{R}(\mathbf{U}, \beta, F_b, T, W, a_T, Q_T, W_f; \theta) \right\|$ 
6:     if  $i = 1$  then
7:        $\alpha^k := 1$  if  $\theta^k > \theta_m^k, k \in [1, n]$ 
8:     end if
9:      $\theta^* \rightarrow (T, W, a_T, Q_T, W_f, \kappa)$ 
10:     $a_n := \|\theta - \theta^*\|_2$ 
11:    if  $i = 1$  then
12:       $r := a_n$ 
13:    else
14:       $r := |a - a_n|$ 
15:    end if
16:     $\theta := \theta^*, i := i + 1; a := a_n$ 
17:  end while
18: end function

```

```

# ensure that we have a steady-state form :
if energy.transient:
    energy.make_steady_state()

# retain base install directory :
out_dir_i = self.out_dir

# directory for saving convergence history :
d_hist = self.out_dir + 'tmc/convergence_history/'
if not os.path.exists(d_hist) and self.MPI_rank == 0:
    os.makedirs(d_hist)

# number of digits for saving variables :
n_i = len(str(max_iter))

# get the bounds of Fb, the max will be updated based on temperate zones :
if energy.energy_flux_mode == 'Fb':
    bounds = copy(wop_kwargs['bounds'])
    self.init_Fb_min(bounds[0], cls=self.this)
    self.init_Fb_max(bounds[1], cls=self.this)
    wop_kwargs['bounds'] = (self.Fb_min, self.Fb_max)

# L_2 error norm between iterations :
abs_error = np.inf
rel_error = np.inf

# number of iterations, from a starting point (useful for restarts) :
if starting_i <= 1:
    counter = 1
else:
    counter = starting_i

# previous velocity for norm calculation
U_prev = self.theta.copy(True)

# perform a fixed-point iteration until the L_2 norm of error
# is less than tolerance :
while abs_error > atol and rel_error > rtol and counter <= max_iter:

    # set a new unique output directory :
    out_dir_n = 'tmc/%0*d/' % (n_i, counter)
    self.set_out_dir(out_dir_i + out_dir_n)

    # solve velocity :
    momentum.solve(annotate=False)

    # update pressure-melting point :
    energy.calc_T_melt(annotate=False)

    # calculate basal friction heat flux :
    momentum.calc_q_fric()

    # derive temperature and temperature-melting flux terms :
    energy.calc_basal_temperature_flux()
    energy.calc_basal_temperature_melting_flux()

    # solve energy steady-state equations to derive temperate zone :
    energy.derive_temperate_zone(annotate=False)

    # fixed-point iteration for thermal parameters and discontinuous
    # properties :
    energy.update_thermal_parameters(annotate=False)

    # calculate the basal-melting rate :
    energy.solve_basal_melt_rate()

    # always initialize Fb to the zero-energy-flux bc :
    Fb_v = self.Mb.vector().array() * self.rhov(0) / self.rhov(0)
    self.init_Fb(Fb_v)

    # update bounds based on temperate zone :
    if energy.energy_flux_mode == 'Fb':
        Fb_m_v = self.Fb_max.vector().array()
        alpha_v = self.alpha.vector().array()
        Fb_m_v[:] = DOLFIN_EPS
        Fb_m_v[alpha_v == 1.0] = bounds[1]
        self.init_Fb_max(Fb_m_v, cls=self.this)

    # optimize the flux of water to remove abnormally high water :
    if energy.energy_flux_mode == 'Fb':
        energy.optimize_water_flux(**wop_kwargs)

    # solve the energy-balance and partition T and W from theta :
    energy.solve(annotate=False)

    # calculate L_2 norms :
    abs_error_n = norm(U_prev.vector() - self.theta.vector(), 'l2')
    tht_nrm = norm(self.theta.vector(), 'l2')

    # save convergence history :
    if counter == 1:
        rel_error = abs_error_n
        if self.MPI_rank == 0:
            err_a = np.array([abs_error_n])
            nrm_a = np.array([tht_nrm])
            np.savetxt(d_hist + 'abs_err.txt', err_a)
            np.savetxt(d_hist + 'theta_norm.txt', nrm_a)
    else:
        rel_error = abs(abs_error - abs_error_n)
        if self.MPI_rank == 0:
            err_n = np.loadtxt(d_hist + 'abs_err.txt')
            nrm_n = np.loadtxt(d_hist + 'theta_norm.txt')
            err_a = np.append(err_n, np.array([abs_error_n]))
            nrm_a = np.append(nrm_n, np.array([tht_nrm]))
            np.savetxt(d_hist + 'abs_err.txt', err_a)
            np.savetxt(d_hist + 'theta_norm.txt', nrm_a)

    # print info to screen :
    if self.MPI_rank == 0:
        s0 = '>>>'
        s1 = 'TMC fixed-point iteration %i (max %i) done: ' % \
            (counter, max_iter)
        s2 = 'r (abs) = %.2e ' % abs_error
        s3 = '(tol %.2e), ' % atol
        s4 = 'r (rel) = %.2e ' % rel_error
        s5 = '(tol %.2e)' % rtol
        s6 = '<<<'
        text0 = get_text(s0, 'red', 1)
        text1 = get_text(s1, 'red')
        text2 = get_text(s2, 'red', 1)
        text3 = get_text(s3, 'red')
        text4 = get_text(s4, 'red', 1)
        text5 = get_text(s5, 'red')
        text6 = get_text(s6, 'red', 1)
        print(text0 + text1 + text2 + text3 + text4 + text5 + text6)

    # update error stuff and increment iteration counter :
    abs_error = abs_error_n
    U_prev = self.theta.copy(True)
    counter += 1

    # call callback function if set :
    if callback != None:
        s = '::: calling thermo-couple-callback function :::'
```

```

print_text(s, cls=self.this)
callback()

# save state to unique hdf5 file :
if isinstance(iter_save_vars, list):
    s = '': saving variables in list arg iter_save_vars :':
    print_text(s, cls=self.this)
    out_file = self.out_dir + 'tmc.h5'
    foutput = HDF5File(mpi_comm_world(), out_file, 'w')
    for var in iter_save_vars:
        self.save_hdf5(var, f=foutput)
    foutput.close()

# reset the base directory ! :
self.set_out_dir(out_dir_i)

# reset the bounds on Fb :
if energy.energy_flux_mode == 'Fb': wop_kwargs['bounds'] = bounds

# save state to unique hdf5 file :
if isinstance(post_tmc_save_vars, list):
    s = '': saving variables in list arg post_tmc_save_vars :':
    print_text(s, cls=self.this)
    out_file = self.out_dir + 'tmc.h5'
    foutput = HDF5File(mpi_comm_world(), out_file, 'w')
    for var in post_tmc_save_vars:
        self.save_hdf5(var, f=foutput)
    foutput.close()

# calculate total time to compute
tf = time()
s = tf - t0
m = s / 60.0
h = m / 60.0
s = s % 60
m = m % 60
text = "time to thermo-couple: %02d:%02d:%02d" % (h,m,s)
print_text(text, 'red', 1)

# plot the convergence history :
s = '': convergence info saved to \'s\' :':
print_text(s % d_hist, cls=self.this)
if self.MPI_rank == 0:
    np.savetxt(d_hist + 'time.txt', np.array([tf - t0]))

err_a = np.loadtxt(d_hist + 'abs_err.txt')
nrm_a = np.loadtxt(d_hist + 'theta_norm.txt')

# plot iteration error :
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_ylabel(r'$\text{Vert } \theta_{n-1} - \theta_n$')
ax.set_xlabel(r'$\text{iteration}$')
ax.plot(err_a, 'k-', lw=2.0)
plt.grid()
plt.savefig(d_hist + 'abs_err.png', dpi=100)
plt.close(fig)

# plot theta norm :
fig = plt.figure()
ax = fig.add_subplot(111)
ax.set_ylabel(r'$\text{Vert } \theta_n$')
ax.set_xlabel(r'$\text{iteration}$')
ax.plot(nrm_a, 'k-', lw=2.0)
plt.grid()
plt.savefig(d_hist + 'theta_norm.png', dpi=100)
plt.close(fig)

```

Code Listing 11.2: CSLVR source code contained in the Enthalpy class used to update the thermal parameters.

```

energy.make_transient(time_step = model.time_step)

def update_thermal_parameters(self, annotate=False):
    """
    fixed-point iterations to make all linearized thermal parameters consistent.
    """
    model = self.model

    # solve the energy equation :
    s = '': updating thermal parameters :':
    print_text(s, cls=self)

    # ensure that we have steady state :
    transient = False
    if self.transient:
        self.make_steady_state()
        transient = True

    # previous theta for norm calculation
    U_prev = self.theta.copy(True)

    # iteration counter :
    counter = 1

    # maximum number of iterations :
    max_iter = 100

    # L2 error norm between iterations :
    abs_error = np.inf
    rel_error = np.inf

    # tolerances for stopping criteria :
    atol = 1e-6
    rtol = 1e-8

    # perform a fixed-point iteration until the L2 norm of error
    # is less than tolerance :
    while abs_error > atol and rel_error > rtol and counter <= max_iter:

        # solve the linear system :
        solve(self.theta_a == self.theta_L, self.theta, self.theta_bc,
              solver_parameters = self.solve_params['solver'], annotate=annotate)

        # solve the non-linear system :
        #model.assign_variable(self.theta, 0.0, annotate=annotate)
        #solve(self.nrg_F == 0, self.theta, J=self.nrg_Jac, bcs=self.theta_bc,
        #      annotate=annotate, solver_parameters=self.solve_params['nparams'])

        # calculate L2 norms :
        abs_error_n = norm(U_prev.vector() - self.theta.vector(), 'l2')
        tht_nrm = norm(self.theta.vector(), 'l2')

        # save convergence history :
        if counter == 1:
            rel_error = abs_error_n
        else:
            rel_error = abs(abs_error - abs_error_n)

        # print info to screen :

```

```

if model.MPI_rank == 0:
    s0 = '':>>>'
    s1 = 'thermal parameter update iteration %i (max %i) done: ' \
        % (counter, max_iter)
    s2 = 'r (abs) = %.2e ' % abs_error
    s3 = '(tol %.2e) ' % atol
    s4 = 'r (rel) = %.2e ' % rel_error
    s5 = '(tol %.2e) ' % rtol
    s6 = ' <<<'
    text0 = get_text(s0, 'red', 1)
    text1 = get_text(s1, 'red')
    text2 = get_text(s2, 'red', 1)
    text3 = get_text(s3, 'red')
    text4 = get_text(s4, 'red', 1)
    text5 = get_text(s5, 'red')
    text6 = get_text(s6, 'red', 1)
    print text0 + text1 + text2 + text3 + text4 + text5 + text6

# update error stuff and increment iteration counter :
abs_error = abs_error_n
U_prev = self.theta.copy(True)
counter += 1

# update the model variable :
model.assign_variable(model.theta, self.theta, annotate=annotate, cls=self)
#model.theta.interpolate(self.theta, annotate=False)
#print_min_max(model.theta, 'theta', cls=self)

# update the temperature and water content for other physics :
self.partition_energy(annotate=annotate)

# derive temperature and temperature-melting flux terms :
self.calc_basal_temperature_flux()
self.calc_basal_temperature_melting_flux()

```

11.1 Plane-strain simulation

For a simple example of TMC Algorithm 4, we use the same plane-strain model as §9.8 using an altered maximum thickness and basal traction. The two-dimensional ice-sheet model uses surface height

$$S(x) = \left(\frac{H_{max} + B_0 - S_0}{2} \right) \cos \left(\frac{2\pi}{\ell} x \right) + \left(\frac{H_{max} + B_0 + S_0}{2} \right),$$

with thickness at the divide H_{max} , height of terminus above water S_0 , depth of ice terminus below water B_0 , and width ℓ . We prescribe the sinusoidally-varying basal topography

$$B(x) = b \cos \left(n_b \frac{2\pi}{\ell} x \right) + B_0,$$

with amplitude b and number of bumps n_b . The basal traction field prescribed follows the surface topography,

$$\beta(x) = \left(\frac{\beta_{max} - \beta_{min}}{2} \right) \cos \left(\frac{2\pi}{\ell} x \right) + \left(\frac{\beta_{max} + \beta_{min}}{2} \right)$$

with maximum value β_{max} and minimum value β_{min} (see Figure 11.1). The surface temperature followed the same sinusoidal pattern as the surface,

$$T_S(x, z) = T_{min} + \lambda_t (H_{max} + B_0 - S_0 - z)$$

with minimum temperature T_{min} and lapse rate λ_t . The specific values used by the simulation are listed in Table 11.1.

The difference between the use of zero-temperate-energy-flux condition (10.44) and water-optimization procedure (10.73) is examined by performing Algorithm 4 using both boundary conditions. Results obtained solving the energy balance with the zero-temperate-energy-flux condition results in a water content in temperate areas reaching unreasonably high levels (Figure 11.4); while the water content field obtained using the F_b -optimization procedure also possessed areas with unreasonably high water content, the situation is much improved (Figure 11.5).

Table 11.1: Plane-strain TMC variables.

Variable	Value	Units	Description
$\dot{\epsilon}_0$	10^{-15}	a^{-1}	strain regularization
F_b	0	m a^{-1}	basal water discharge
k_x	150	—	number of x divisions
k_z	50	—	number of z divisions
N_e	15000	—	number of cells
N_n	7701	—	number of vertices
ℓ	400	km	width of domain
H_{max}	3000	m	thickness at divide
S_0	100	m	terminus height
B_0	-200	m	terminus depth
n_b	25	—	number of bed bumps
b	50	m	bed bump amplitude
β_{max}	1000	$\text{kg m}^{-2}\text{a}^{-1}$	max basal traction
β_{min}	100	$\text{kg m}^{-2}\text{a}^{-1}$	min basal traction
T_{min}	228.15	K	min. temperature
λ_t	$6.5\text{e}-3$	K m^{-1}	lapse rate
W_c	0.03	—	maximum basal W
k_0	10^{-3}	—	non-adv. flux coef.
q_{geo}	4.2×10^{-2}	W m^{-2}	geothermal heat flux

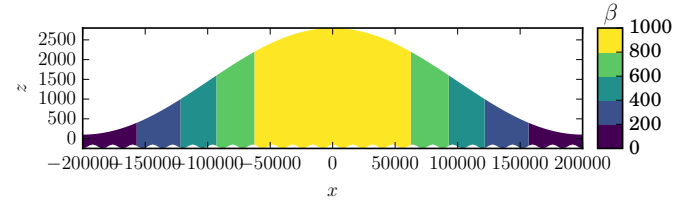
The convergence behavior associated with the optimization-procedure simulation appears to become unstable around iterate 200 (Figure 11.2). This may be due to the extreme nature of the simulation, or an indication that the optimization procedure of §10.6 may require further constraints on basal water discharge F_b . Note that the norm of the current energy guess using a zero basal-energy flux is much larger than that obtained using the F_b -optimization procedure (Figure 11.3). This is because the water content using this method is allowed to reach values associated with ice composed of up to $\approx 50\%$ water, an extremely unlikely event. Additionally, note that the use of W_f in flow-rate factor (10.22) prevents rate-factor A from reaching levels beyond empirical evidence, and also that if we had removed any energy above $\theta_c = \theta + W_c L$ as in Greve and Blatter (2009), or through a time-dependent function similar to that used by Aschwanden et al. (2012), the values of $\|\theta_n\|_2$ would be closer to that obtained by our F_b -optimization procedure in Figure 11.3. Finally, due to the fact that zero-basal-energy flux boundary (10.44) only removes water generated at the basal surface, the unadjusted internal water content values quantify the amount of water generated by strain-heating within the ice.

The CSLVR script used to solve this problem is shown in Code Listing 11.3, and the code used to generate Figures 11.4 and 11.5 in Code Listing 11.4.

Code Listing 11.3: CSLVR script which performs the plane-strain TMC simulation of §11.1.

```
from csivr import *

# this problem has a few more constants than the last :
l = 400000 # width of the domain
Hmax = 3000 # thickness at the divide
S0 = 100 # terminus height above water
B0 = -200 # terminus depth below water
nb = 25 # number of basal bumps
b = 50 # amplitude of basal bumps
betaMax = 1000 # maximum traction coefficient
betaMin = 100 # minimum traction coefficient
Tmin = 273.15 - 45 # temperature at divide
```

Figure 11.1: The basal traction β used for the TMC-simulation.

```
St = 6.5 / 1000.0 # lapse rate

# create the rectangle mesh, to be deformed by the chosen geometry later :
p1 = Point(-1/2, 0.0) # lower-left corner
p2 = Point(1/2, 1.0) # upper-right corner
kx = 150 # number of x-divisions
kz = 50 # number of z-divisions
mesh = RectangleMesh(p1, p2, kx, kz)

# these control the basal-boundary condition :
e_mode = 'zero_energy'
e_mode = 'Fb'

# the output directories :
out_dir = 'ps_results_new/' + e_mode + '/'
plt_dir = '../images/tmc/plane_strain_new/' + e_mode + '/'

# this is a lateral mesh problem, defined in the x,z plane. Here we use
# linear-Lagrange elements corresponding with order=1 :
model = LatModel(mesh, out_dir=out_dir, order=1)

# the expressions for our data :
S = Expression('(Hmax+B0-S0)/2*cos(2*pi*x[0]/l) + (Hmax+B0+S0)/2',
               Hmax=Hmax, B0=B0, S0=S0, l=l,
               element = model.Q.ufl_element())
B = Expression('(b*cos(nb*2*pi*x[0]/l) + B0)',
               b=b, l=l, B0=B0, nb=nb,
               element = model.Q.ufl_element())
b = Expression('(bMax - bMin)/2.0*cos(2*pi*x[0]/l) + (bMax + bMin)/2.0',
               bMax=betaMax, bMin=betaMin, l=l,
               element = model.Q.ufl_element())
T = Expression('(Tmin + St*(Hmax + B0 - S0 - x[1]))',
               Tmin=Tmin, Hmax=Hmax, B0=B0, S0=S0, St=St,
               element = model.Q.ufl_element())

# deform the geometry to match the surface and bed functions :
model.deform_mesh_to_geometry(S, B)

# mark the facets and cells for proper integration :
model.calculate_boundaries(mask=None)

# initialize the variables :
model.init_beta(b) # traction
model.init_T(T) # internal temperature
model.init_T_surface(T) # atmospheric temperature
model.init_Wc(0.03) # max basal water content
model.init_k_0(5e-3) # non-advective flux coef.
model.init_q_geo(model.ghf) # geothermal flux
model.solve_hydrostatic_pressure() # for pressure-melting
model.form_energy_dependent_rate_factor() # thermo-mech coupling

# the momentum and energy physics :
mom = MomentumDukowiczPlaneStrain(model)

# the energy physics using the chosen basal energy flux mode and
# Galerkin/least-squares stabilization :
nrg = Enthalpy(model, mom, energy_flux_mode = e_mode,
               stabilization_method = 'GLS')

# thermo-solve callback function, at the end of each TMC iteration :
def tmc_cb_ftn():
    nrg.calc_PE() # calculate grid Peclet number
    nrg.calc_vert_avg_strain_heat() # calc vert. avg. of Q
    nrg.calc_vert_avg_W() # calc vert. avg. of W
    nrg.calc_temp_rat() # calc ratio of H that is temperate

# at the end of the TMC procedure, save the state of these functions :
tmc_save_vars = [model.T,
                 model.W,
                 model.Fb,
                 model.Mb,
                 model.alpha,
                 model.alpha_int,
                 model.PE,
                 model.Wbar,
                 model.Qbar,
                 model.temp_rat,
                 model.US,
                 model.p,
                 model.beta,
                 model.theta]

# form the objective functional for water-flux optimization :
nrg.form_cost_ftn(kind='L2')

# the water-content optimization problem args :
wop_kwargs = {'max_iter' : 25,
              'bounds' : (DOLFIN_EPS, 100.0),
              'method' : 'ipopt',
              'adj_callback' : None}

# thermo-mechanical coupling args :
tmc_kwargs = {'momentum' : mom,
              'energy' : nrg,
              'wop_kwargs' : wop_kwargs,
              'callback' : tmc_cb_ftn,
              'atol' : 1e2,
              'rtol' : 1e0,
              'max_iter' : 20,
              'iter_save_vars' : None,
              'post_tmc_save_vars' : tmc_save_vars,
              'starting_i' : 1}

# thermo_solve :
model.thermo_solve(**tmc_kwargs)
```



```
# save the mesh for plotting the data saved by tmc_save_vars :
f = HDF5File(mpi_comm_world(), out_dir + 'state.h5', 'w')
model.save_subdomain_data(f)
model.save_mesh(f)
f.close()
```

```
# plotting :
```

```
figsize = (10,2.2)
```

```
model.init_U_mag(model.U3)
U_min = model.U_mag.vector().min()
U_max = model.U_mag.vector().max()
U_lvls = array([U_min, 1e2, 1e3, 1e4, 1.5e4, U_max])
plot_variable(u = model.U_mag, name = 'U_mag', direc = plt_dir,
    figsize = figsize,
    title = r'$\text{Vert } \mathbf{u} \text{ } \text{Vert}$',
    cmap = 'viridis',
    levels = None, #U_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
p_min = model.p.vector().min()
p_max = model.p.vector().max()
p_lvls = array([p_min, 1e6, 5e6, 1e7, 1.5e7, 2e7, 2.5e7, p_max])
plot_variable(u = model.p, name = 'p', direc = plt_dir,
    figsize = figsize,
    title = r'$p$',
    cmap = 'viridis',
    levels = None, #p_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
beta_lvls = array([0, 200, 400, 600, 800, 1000])
plot_variable(u = model.beta, name = 'beta', direc = plt_dir,
    figsize = (6,2),
    title = r'$\beta$',
    cmap = 'viridis',
    levels = None, #beta_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    cb_format = '%g')
```

```
T_min = model.T.vector().min()
T_max = model.T.vector().max()
T_lvls = array([T_min, 230, 240, 250, 260, 265, 270, T_max])
plot_variable(u = model.T, name = 'T', direc = plt_dir,
    figsize = figsize,
    title = r'$T$',
    cmap = 'viridis',
    levels = None, #T_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

```
W_min = model.W.vector().min()
W_max = model.W.vector().max()
W_lvls = array([0.0, 1e-2, 5e-2, W_max])
plot_variable(u = model.W, name = 'W', direc = plt_dir,
    figsize = figsize,
    title = r'$W$',
    cmap = 'viridis',
    levels = None, #W_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
plot_variable(u = model.theta, name = 'theta', direc = plt_dir,
    figsize = figsize,
    title = r'$\theta$',
    cmap = 'viridis',
    levels = None,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%g')
```

```
Mb_min = model.Mb.vector().min()
Mb_max = model.Mb.vector().max()
Mb_lvls = array([0.0, 0.2, 0.3, 0.4, 0.5, Mb_max])
plot_variable(u = model.Mb, name = 'Mb', direc = plt_dir,
    figsize = figsize,
    title = r'$M_b$',
    cmap = 'viridis', #gist_yarg',
    levels = None, #Mb_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

```
Fb_min = model.Fb.vector().min()
Fb_max = model.Fb.vector().max()
Fb_lvls = array([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, Fb_max])
plot_variable(u = model.Fb, name = 'Fb', direc = plt_dir,
    figsize = figsize,
    title = r'$F_b$',
    cmap = 'viridis',
    levels = None, #Fb_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

```
model = LatModel(fstate, out_dir = out_dir, use_periodic = False)
model.set_subdomains(fstate)
```

```
model.init_T(f)
model.init_W(f)
model.init_Fb(f)
model.init_Mb(f)
model.init_alpha(f)
model.init_alpha_int(f)
model.init_PE(f)
model.init_Wbar(f)
model.init_Qbar(f)
model.init_temp_rat(f)
model.init_U(f)
model.init_p(f)
model.init_beta(f)
model.init_theta(f)
```

```
# plotting :
```

```
figsize = (10,2.2)
```

```
U_min = model.U_mag.vector().min()
U_max = model.U_mag.vector().max()
U_lvls = array([U_min, 100, 200, 400, 600, U_max])
plot_variable(u = model.U_mag, name = 'U_mag', direc = plt_dir,
    figsize = figsize,
    title = r'$\text{Vert } \mathbf{u} \text{ } \text{Vert}$',
    cmap = 'viridis',
    levels = U_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
p_min = model.p.vector().min()
p_max = model.p.vector().max()
p_lvls = array([p_min, 1e6, 5e6, 1e7, 1.5e7, 2e7, 2.5e7, p_max])
plot_variable(u = model.p, name = 'p', direc = plt_dir,
    figsize = figsize,
    title = r'$p$',
    cmap = 'viridis',
    levels = p_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
beta_lvls = array([0, 200, 400, 600, 800, 1000])
plot_variable(u = model.beta, name = 'beta', direc = plt_dir,
    figsize = (6,2),
    title = r'$\beta$',
    cmap = 'viridis',
    levels = beta_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    cb_format = '%g')
```

```
T_min = model.T.vector().min()
T_max = model.T.vector().max()
T_lvls = array([T_min, 230, 240, 250, 260, 265, 270, T_max])
plot_variable(u = model.T, name = 'T', direc = plt_dir,
    figsize = figsize,
    title = r'$T$',
    cmap = 'viridis',
    levels = T_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

```
W_min = model.W.vector().min()
W_max = model.W.vector().max()
W_lvls = array([0.0, 1e-2, 5e-2, W_max])
plot_variable(u = model.W, name = 'W', direc = plt_dir,
    figsize = figsize,
    title = r'$W$',
    cmap = 'viridis',
    levels = W_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1e')
```

```
plot_variable(u = model.theta, name = 'theta', direc = plt_dir,
    figsize = figsize,
    title = r'$\theta$',
    cmap = 'viridis',
    levels = None,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%g')
```

```
Mb_min = model.Mb.vector().min()
Mb_max = model.Mb.vector().max()
Mb_lvls = array([0.0, 0.2, 0.3, 0.4, 0.5, Mb_max])
plot_variable(u = model.Mb, name = 'Mb', direc = plt_dir,
    figsize = figsize,
    title = r'$M_b$',
    cmap = 'viridis', #gist_yarg',
    levels = Mb_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

```
Fb_min = model.Fb.vector().min()
Fb_max = model.Fb.vector().max()
Fb_lvls = array([0.0, 0.1, 0.2, 0.3, 0.4, 0.5, Fb_max])
plot_variable(u = model.Fb, name = 'Fb', direc = plt_dir,
    figsize = figsize,
    title = r'$F_b$',
    cmap = 'viridis',
    levels = Fb_lvls,
    show = False,
    ylabel = r'$z$',
    equal_axes = False,
    extend = 'both',
    cb_format = '%.1f')
```

Code Listing 11.4: CSLVR script which plots the result generated by Code Listing 11.3.

```
from csivr import *

#e_mode = 'zero_energy'
e_mode = 'Fb'
out_dir = 'ps_results/' + e_mode + '/'
plt_dir = './../images/tmc/plane_strain/' + e_mode + '/'

f = HDF5File(mpi_comm_world(), out_dir + 'tmc.h5', 'r')
fstate = HDF5File(mpi_comm_world(), out_dir + 'state.h5', 'r')
```

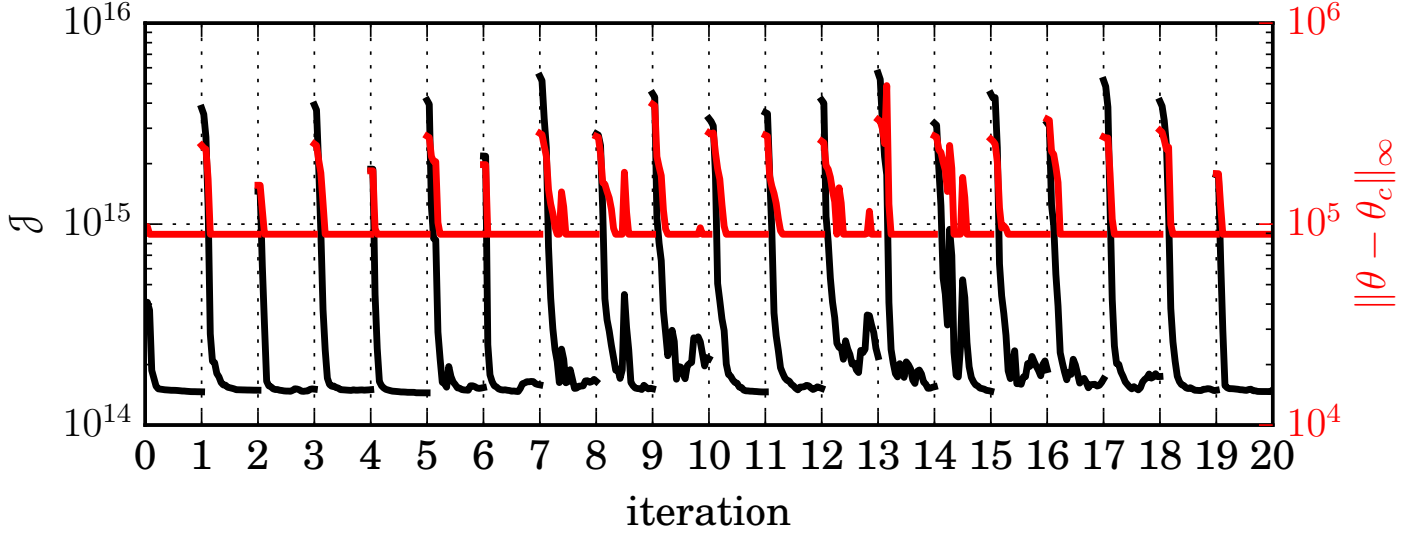


Figure 11.2: Objective functional values (10.72) for each iteration of Algorithm 4 (left axis, black) and misfit between the critical value of current energy value θ and the critical energy value $\theta_c = \theta + W_c L$ (right axis, red) for the F_b -optimization procedure of §10.6.

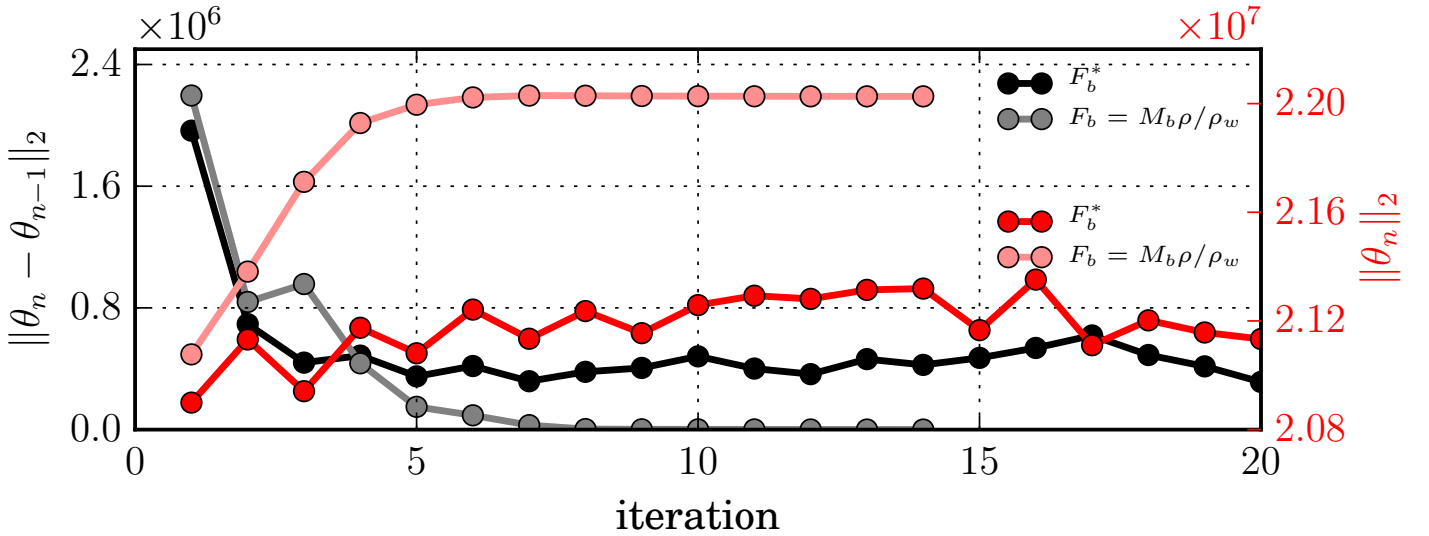


Figure 11.3: Convergence plot of TMC algorithm 4 applied to the plane-strain simulation using both the zero-energy-basal-boundary condition (10.44) corresponding with $F_b = M_b \rho / \rho_w$ (left axis, grey), and the F_b -optimization procedure of §10.6 using basal-water-discharge-boundary condition (10.43) (left axis, black). Also shown is the norm of the current energy guess θ_n for the zero-energy-basal-boundary condition simulation (right axis, pink) and the F_b -optimization procedure simulation (right axis, red).

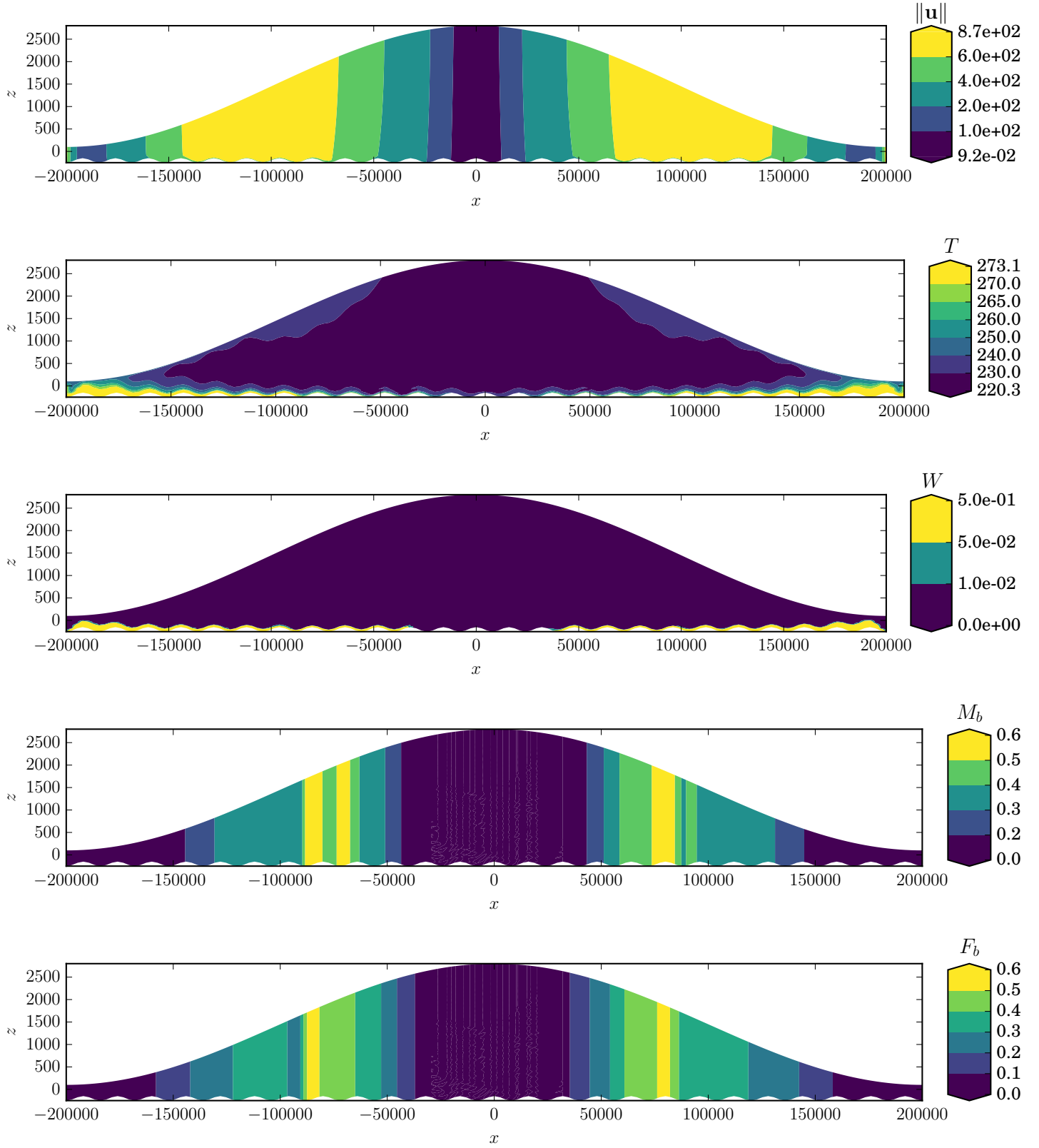


Figure 11.4: The plane-strain results attained using zero-temperature-energy-flux boundary condition (10.44). From top to bottom: velocity magnitude $\|\mathbf{u}\|$, temperature T , water content W , basal melt rate M_b , and basal water discharge F_b .

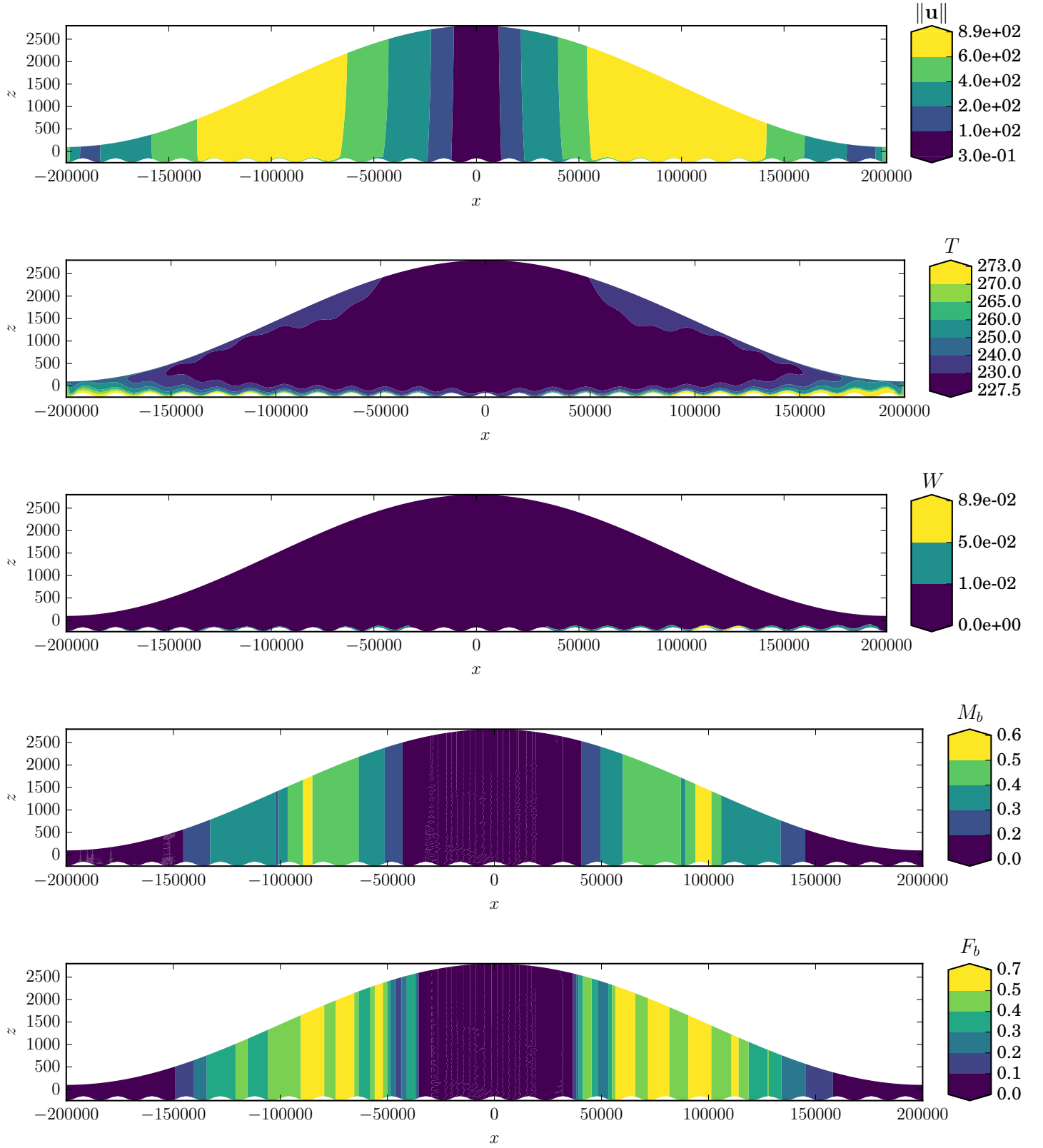


Figure 11.5: Plane-strain results attained using the F_b optimization process of §10.6 with boundary condition (10.43). From top to bottom: velocity magnitude $\|\mathbf{u}\|$, temperature T , water content W , basal melt rate M_b , and basal water discharge F_b .

Chapter 12

Inclusion of velocity data

In order to partition the effects of energy-enhancement and traction-diminishment on momentum – and thus ensure consistency between rate factor $A(T, W)$ given by (10.22) and traction coefficient β in tangential stress condition (9.3), (9.30), (9.43), and (9.57) – an optimization procedure for basal traction is performed for a previously attained energy θ . Because shear viscosity (8.10), (9.22), (9.38) and (9.53) decreases for increasing energy, the ice speeds up for higher temperature T and water content W due to increased deformation. Similarly, larger values of basal-traction β decrease the vertical average of the mixture velocity while possibly generating more energy due to frictional- and strain-heating. Indubitably, any generated heat resulting from these processes induces deformation, and therefore also increases the mixture velocity. Hence a feedback process exists which requires extra care be taken in order to constrain both energy and traction.

An energy constraint already exists as evident by coefficient W_f in rate factor (10.22); due to lack of empirical evidence, enhancement is limited for ice containing water contents in excess of 1%. Regarding basal traction β , it is desirable to penalize abnormally high spatial gradients in order to reduce non-physical oscillations (Vogel, 2002).

The process used to optimize traction coefficient β – referred to in this context as *data assimilation* – is directly analogous to the process used to optimize the basal-water content and energy previously described; a momentum objective functional $\mathcal{J}(\mathbf{u}_h, \beta) : \mathcal{H}^1(\Omega) \times \mathcal{H}^1(\Omega) \rightarrow \mathbb{R}$ is minimized over the domain of the ice-sheet Ω . This objective functional measures the misfit between the observed velocities $\mathbf{u}_{ob} = [u_{ob} \ v_{ob}]^T$ and modeled velocities $\mathbf{u}_h = [u \ v]^T$ over upper ice-sheet surface Γ_S ,

$$\mathcal{J}(\mathbf{u}_h, \beta) = \gamma_1 \mathcal{J}_1(\mathbf{u}_h) + \gamma_2 \mathcal{J}_2(\mathbf{u}_h) + \gamma_3 \mathcal{J}_3(\beta) + \gamma_4 \mathcal{J}_4(\beta), \quad (12.1)$$

where

$$\mathcal{J}_1(\mathbf{u}_h) = \frac{1}{2} \int_{\Gamma_S} [(u - u_{ob})^2 + (v - v_{ob})^2] d\Gamma_S \quad (12.2)$$

$$\mathcal{J}_2(\mathbf{u}_h) = \frac{1}{2} \int_{\Gamma_S} \ln \left(\frac{(u^2 + v^2)^{1/2} + u_0}{(u_{ob}^2 + v_{ob}^2)^{1/2} + u_0} \right)^2 d\Gamma_S \quad (12.3)$$

$$\mathcal{J}_3(\beta) = \frac{1}{2} \int_{\Gamma_G} \nabla \beta \cdot \nabla \beta d\Gamma_G \quad (12.4)$$

$$\mathcal{J}_4(\beta) = \int_{\Gamma_G} (\nabla \beta \cdot \nabla \beta + \beta_0)^{1/2} d\Gamma_G, \quad (12.5)$$

with L^2 cost coefficient γ_1 , *logarithmic* cost coefficient γ_2 , *Tikhonov* regularization parameter γ_3 , and *total variation* (TV) regularization parameter γ_4 . Here, $u_0 = 10^{-2}$ and $\beta_0 = 10^{-16}$ terms are added to avoid singularities. Note that the functionals (12.2) and (12.3) are referred to as *cost functionals* while (12.4) and (12.5) are referred to as *regularization functionals*.

By forming this objective with cost functionals stated in terms of both L^2 and logarithmic velocity misfit terms, priority is given to either fast or slow areas of flow by adjusting the values of γ_1 and γ_2 , respectively (Morlighem, Seroussi, and Rignot, 2013). Note that larger values of γ_3 and γ_4 result in increased regularity of β , at the cost of increased misfit $\|\mathbf{u}_h - \mathbf{u}_{ob}\|$, thus weighting the associated gradient penalty functionals in relation to the other functionals in (12.1). Finally, the addition of the total variation functional (12.5) in objective (12.1) reduces short-wavelength oscillations that only marginally affect Tikhonov regularization functional (12.4).

12.1 Momentum optimization procedure

One of the advantages of the action principles presented by Dukowicz, Price, and Lipscomb (2010) is that actions (9.8), (9.33), (9.45), and (9.62) are all self-adjoint. Indeed, the momentum Lagrangian functional associated with momentum objective (12.1) and first-order action principle (9.33) – used in an analogous set of KKT conditions for momentum as general KKT condition (7.13) – is defined as

$$\mathcal{H}(\mathbf{u}_h, \beta, \boldsymbol{\lambda}) = \mathcal{J}(\mathbf{u}_h, \beta) + \delta_{\mathbf{u}_h} \mathcal{A}_{BP}(\boldsymbol{\lambda}), \quad (12.6)$$

with momentum adjoint variable $\boldsymbol{\lambda} = [\lambda_x \ \lambda_y]^T \in (\mathcal{H}^1(\Omega))^2$.

The log-barrier problem (see §7.2) for momentum is of the same form as energy barrier problem (10.80). Thus, the associated minimization problem for control parameter β and state parameter \mathbf{u}_h is

$$\min_{\beta} \left\{ \varphi_{\omega}(\mathbf{u}_h, \beta) = \mathcal{J}(\mathbf{u}_h, \beta) - \omega \sum_{i=1}^n \ln(\beta_i) \right\} \quad (12.7)$$

for a decreasing sequence of barrier parameters ω converging to zero, and n is the number of quadrature points in the discretization. The logarithmic sum term in (12.7) ensures that

β remains positive, as required by tangential basal stress condition (9.3).

Note that I have used first-order action (9.33) in Lagrangian (12.6); it has been my experience that solving full-Stokes momentum balance (9.13) using a coarse mesh (≈ 1 km minimum cell diameter) and traction β^* resulting from the minimization of first-order momentum barrier problem (12.7) results in a velocity field that differs only slightly from that obtained using first-order momentum balance (9.32). This simplification provides a several-fold improvement in computation time. Note also that the reformulated-Stokes action principle presented in §9.4 presents a substantial improvement in the velocity approximation from the first-order model at higher basal gradients (see §9.7). However, at its current state of development, CSLVR has not incorporated this model with the automated adjoint-optimization software it currently employs, Dofin-Adjoint.

Additional computational energy is saved by using a linearization of rate-factor (10.22) derived from a previously thermo-mechanically coupled \mathbf{u}_h^{i-1} . This converts first-order viscosity (9.22) into

$$\eta_{BP}^L(\theta, \mathbf{u}_h^{i-1}) = \frac{1}{2} A(\theta)^{-1/n} (\dot{\epsilon}_{BP}(\mathbf{u}_h^{i-1}) + \dot{\epsilon}_0)^{\frac{1-n}{n}}, \quad (12.8)$$

and first-order-viscous-dissipation term (9.34) in (9.33) into

$$V^L(\dot{\epsilon}_{BP}^2) = \int_0^{\dot{\epsilon}_{BP}^2} \eta_{BP}^L(s) ds = \eta_{BP}^L(\theta, \mathbf{u}_h^{i-1}) \dot{\epsilon}_{BP}^2. \quad (12.9)$$

Finally, $\delta_{\mathbf{u}_h} \mathcal{A}_{BP}(\lambda)$ in Lagrangian (12.6) is as derived by Dukowicz, Price, and Lipscomb (2010),

$$\begin{aligned} \mathcal{H}(\mathbf{u}_h, \beta, \lambda) = & \mathcal{J}(\mathbf{u}_h, \beta) + \int_{\Gamma_E} f_e \mathbf{n}_h \cdot \boldsymbol{\lambda} d\Gamma_E \\ & + \int_{\Omega} \sigma_{BP}^L : \nabla \boldsymbol{\lambda} d\Omega - \int_{\Omega} \rho g (\nabla S)_h \cdot \boldsymbol{\lambda} d\Omega \\ & + \int_{\Gamma_B} (\Lambda_{BP}^L \boldsymbol{\lambda} \cdot \mathbf{n}_h + \beta \mathbf{u}_h \cdot \boldsymbol{\lambda}) d\Gamma_B, \end{aligned} \quad (12.10)$$

where σ_{BP}^L and Λ_{BP}^L are the linearized counterparts to first-order quasi-stress tensor (9.26) and impenetrability Lagrange multiplier (9.35) utilizing linear viscosity (12.8).

The CSLVR source code implementation of this procedure is shown in Code Listing 12.1.

Code Listing 12.1: CSLVR source code of the abstract class Momentum for optimizing the velocity and traction. All of the momentum models of §9 inherit this method.

```
def optimize_U_ob(self, control, bounds,
    method = 'l_bfgs-b',
    max_iter = 100,
    adj_save_vars = None,
    adj_callback = None,
    post_adj_callback = None):
    """
    s = """ solving optimal control to minimize ||u - u_ob|| with " + \
        "control parameter '%s' """
    print_text(s % control.name(), cls=self)
    model = self.model

    # reset entire dolfin-adjoint state :
    adj_reset()

    # starting time :
    t0 = time()

    # need this for the derivative callback :
    global counter
```

```
counter = 0

# functional lists to be populated :
global Rs, Js, Ds, J1s, J2s, R1s, R2s
Rs = []
Js = []
Ds = []
if self.obj_ftn_type == 'log_L2_hybrid':
    J1s = []
    J2s = []
if self.reg_ftn_type == 'TV_Tik_hybrid':
    R1s = []
    R2s = []

# solve the momentum equations with annotation enabled :
s = """ solving momentum forward problem """
print_text(s, cls=self)
self.solve(annotate=True)

# now solve the control optimization problem :
s = """ starting adjoint-control optimization with method '%s' """
print_text(s % method, cls=self)

# objective function callback function :
def eval_cb(I, c):
    s = """ adjoint objective eval post callback function """
    print_text(s, cls=self)
    print_min_max(I, 'I', cls=self)
    print_min_max(c, 'control', cls=self)

# objective gradient callback function :
def deriv_cb(I, dI, c):
    global counter, Rs, Js, J1s, J2s
    if method == 'ipopt':
        s1 = '>>>'
        s2 = '<<<'
        text0 = get_text(s0, 'red', 1)
        text1 = get_text(s1 % (counter, max_iter), 'red')
        text2 = get_text(s2, 'red', 1)
        if MPI.rank(mpi_comm_world())==0:
            print(text0 + text1 + text2)
        counter += 1
    s = """ adjoint obj. gradient post callback function """
    print_text(s, cls=self)
    print_min_max(dI, 'dI/dcontrol', cls=self)

# update the DA current velocity to the model for evaluation
# purposes only; the model.assign_variable function is
# annotated for purposes of linking physics models to the adjoint
# process :
u_opt = DofinAdjointVariable(model.U3).tape_value()
model.init_U(u_opt, cls=self)

# print functional values :
control.assign(c, annotate=False)
ftnls = self.calc_functionals()
D = self.calc_misfit()

# functional lists to be populated :
Rs.append(ftnls[0])
Js.append(ftnls[1])
Ds.append(D)
if self.obj_ftn_type == 'log_L2_hybrid':
    J1s.append(ftnls[2])
    J2s.append(ftnls[3])
if self.reg_ftn_type == 'TV_Tik_hybrid':
    R1s.append(ftnls[4])
    R2s.append(ftnls[5])

# call that callback, if you want :
if adj_callback is not None:
    adj_callback(I, dI, c)

# get the cost, regularization, and objective functionals :
I = self.J + self.R

# define the control parameter :
m = Control(control, value=control)

# create the reduced functional to minimize :
F = ReducedFunctional(Functional(I), m, eval_cb=eval_cb,
    derivative_cb=deriv_cb)

# optimize with scipy's fmin_l_bfgs_b :
if method == 'l_bfgs-b':
    out = minimize(F, method="L-BFGS-B", tol=1e-9, bounds=bounds,
        options={"disp": True,
            "maxiter": max_iter,
            "gtol": 1e-5})
    b_opt = out

# or optimize with IPOPT (preferred) :
elif method == 'ipopt':
    try:
        import pyipopt
    except ImportError:
        info_red("""You do not have IPOPT and/or pyipopt installed.
            When compiling IPOPT, make sure to link against HSL,
            as it is a necessity for practical problems.""")
        raise
    problem = MinimizationProblem(F, bounds=bounds)
    parameters = {"tol": 1e-8,
        "acceptable_tol": 1e-6,
        "maximum_iterations": max_iter,
        "print_level": 5,
        "ma97_order": "metis",
        "linear_solver": "ma97"}
    solver = IPOPTSolver(problem, parameters=parameters)
    b_opt = solver.solve()

# make the optimal control parameter available :
model.assign_variable(control, b_opt, cls=self)
#Control(control).update(b_opt) # FIXME: does this work?

# call the post-adjoint callback function if set :
if post_adj_callback is not None:
    s = """ calling optimize_U_ob() post-adjointed callback function """
    print_text(s, cls=self)
    post_adj_callback()

# save state to unique hdf5 file :
if isinstance(adj_save_vars, list):
    s = """ saving variables in list arg adj_save_vars """
    print_text(s, cls=self)
    out_file = model.out_dir + 'u_opt.h5'
    foutput = HDF5File(mpi_comm_world(), out_file, 'w')
    for var in adj_save_vars:
        model.save_hdf5(var, f=foutput)
    foutput.close()

# calculate total time to compute
tf = time()
```

```

s = tf - t0
m = s / 60.0
h = m / 60.0
s = s % 60
m = m % 60
text = "time to optimize |u - u_ob||: %02d:%02d:%02d" % (h,m,s)
print_text(text, 'red', 1)

# save all the objective functional values with rudimentary plot :
d = model.out_dir + 'objective_ftnls_history/'
s = '...' saving objective functionals to %s ...'
print_text(s % d, cls=self)
if model.MPI_rank==0:
    if not os.path.exists(d):
        os.makedirs(d)
    np.savetxt(d + 'time.txt', np.array([tf - t0]))
    np.savetxt(d + 'Rs.txt', np.array(Rs))
    np.savetxt(d + 'Js.txt', np.array(Js))
    np.savetxt(d + 'Ds.txt', np.array(Ds))
    if self.obj_ftn_type == 'log_L2_hybrid':
        np.savetxt(d + 'J1s.txt', np.array(J1s))
        np.savetxt(d + 'J2s.txt', np.array(J2s))
    if self.reg_ftn_type == 'TV_Tik_hybrid':
        np.savetxt(d + 'R1s.txt', np.array(R1s))
        np.savetxt(d + 'R2s.txt', np.array(R2s))

fig = plt.figure()
ax = fig.add_subplot(111)
#ax.set_yscale('log')
ax.set_ylabel(r'$\mathscr{J}$\left( \mathbf{u} \right)$')
ax.set_xlabel(r'iteration')
ax.plot(np.array(Js), 'r-', lw=2.0)
plt.grid()
plt.savefig(d + 'J.png', dpi=100)
plt.close(fig)

fig = plt.figure()
ax = fig.add_subplot(111)
#ax.set_yscale('log')
ax.set_ylabel(r'$\mathscr{R}$\left( \beta \right)$')
ax.set_xlabel(r'iteration')
ax.plot(np.array(Rs), 'r-', lw=2.0)
plt.grid()
plt.savefig(d + 'R.png', dpi=100)
plt.close(fig)

fig = plt.figure()
ax = fig.add_subplot(111)
#ax.set_yscale('log')
ax.set_ylabel(r'$\mathscr{D}$\left( \mathbf{u} \right)$')
ax.set_xlabel(r'iteration')
ax.plot(np.array(Ds), 'r-', lw=2.0)
plt.grid()
plt.savefig(d + 'D.png', dpi=100)
plt.close(fig)

if self.obj_ftn_type == 'log_L2_hybrid':
    fig = plt.figure()
    ax = fig.add_subplot(111)
    #ax.set_yscale('log')
    ax.set_ylabel(r'$\mathscr{J}_1$')
    ax.set_xlabel(r'iteration')
    ax.plot(np.array(J1s), 'r-', lw=2.0)
    plt.grid()
    plt.savefig(d + 'J1.png', dpi=100)
    plt.close(fig)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    #ax.set_yscale('log')
    ax.set_ylabel(r'$\mathscr{J}_2$')
    ax.set_xlabel(r'iteration')
    ax.plot(np.array(J2s), 'r-', lw=2.0)
    plt.grid()
    plt.savefig(d + 'J2.png', dpi=100)
    plt.close(fig)

if self.reg_ftn_type == 'TV_Tik_hybrid':
    fig = plt.figure()
    ax = fig.add_subplot(111)
    #ax.set_yscale('log')
    ax.set_ylabel(r'$\mathscr{T}$\left( \beta \right)$')
    ax.set_xlabel(r'iteration')
    ax.plot(np.array(R1s), 'r-', lw=2.0)
    plt.grid()
    plt.savefig(d + 'R1.png', dpi=100)
    plt.close(fig)

    fig = plt.figure()
    ax = fig.add_subplot(111)
    #ax.set_yscale('log')
    ax.set_ylabel(r'$\mathscr{T}$\left( \beta \right)$')
    ax.set_xlabel(r'iteration')
    ax.plot(np.array(R2s), 'r-', lw=2.0)
    plt.grid()
    plt.savefig(d + 'R2.png', dpi=100)
    plt.close(fig)

```

12.2 Dual optimization for energy and momentum

To begin the procedure of solving energy and momentum, the thermo-mechanical coupling process described in Algorithm 4 is performed for an initial friction field β^i , energy θ^i , and basal water discharge F_b^i . After this, barrier problem (12.7) is repeatedly solved until the difference between two subsequent traction fields are below a specified tolerance. This procedure is outlined by Algorithm 6 with CSLVR implementation shown in Code Listing 12.2.

As suggested by Morlighem, Seroussi, and Rignot (2013), a

suitable initialization of the traction field β^i may be formed by vertically integrating first-order momentum balance (9.25) and eliminating any horizontal derivatives – i.e., longitudinal stretching and lateral shearing – from the left-hand side, resulting in

$$\beta \mathbf{u}_h|_B = \rho g H (\nabla S)_h, \quad (12.11)$$

where $H = S - B$ is the ice thickness and $\beta \mathbf{u}_h|_B$ is the entire contribution of vertical shear, referred to as *basal traction*. Note that this derivation of the momentum balance is equivalent to the *shallow-ice approximation* for glacier flow (Greve and Blatter, 2009). Finally, using the observed surface velocity as an approximation for basal velocity $\mathbf{u}_h|_B$ and taking the norm of vector expression (12.11), the shallow-ice-approximate traction field is derived,

$$\beta_{\text{SIA}} = \frac{\rho g H \|(\nabla S)_h\|}{\|\mathbf{u}_{ob}\| + u_0}, \quad (12.12)$$

where u_0 is a small positive speed to avoid singularities. Notice that in areas without surface velocity observations, the approximate traction field (12.12) will be invalid. Therefore, for the purposes of creating an initial traction field, we replace any areas containing missing measurements of velocity \mathbf{u}_{ob} in (12.12) with the balance velocity $\bar{\mathbf{u}}$ as derived in Chapter 13.

It has been my experience that traction β will converge consistently if initialized to the same value prior to solving system (12.7). Therefore, at the start of every iteration, β is reset to its initial value β^i . See Algorithm 6 for details, and Chapter 16 for an example of the full energy and momentum optimization procedure applied to the region of Greenland's Jakobshavn Glacier. In the example that follows, we solve an isothermal momentum-optimization problem.

Algorithm 6 – TMC basal-friction data assimilation

```

1: function TMC_DA( $\beta^i, \theta^i, F_b^i, n_{\max}$ )
2:    $a_{tol} := 100$ 
3:    $r := \infty$ 
4:    $i := 1$ 
5:    $\beta^p := \beta^i$ 
6:    $\theta, F_b := \text{TMC}(\beta^i, \theta^i, F_b^i)$ 
7:   while  $r > a_{tol}$  and  $i < n_{\max}$  do
8:      $\beta := \beta^i$ 
9:      $\beta^* := \underset{\beta}{\operatorname{argmin}} \{ \varphi_{\omega}(\beta) \}$ 
10:     $\theta, F_b := \text{TMC}(\beta^*, \theta, F_b)$ 
11:     $r := \|\beta^p - \beta^*\|_2$ 
12:     $\beta^p := \beta^*$ 
13:     $i := i + 1$ 
14:   end while
15:   return  $\beta^*$ 
16: end function

```

Code Listing 12.2: Implementation of Algorithm 6 by CSLVR contained in the Model class for the optimization of a Momentum and Energy instance.


```

def assimilate_U_ob(self, momentum, beta_i, max_iter,
                    tmc_kwargs, uop_kwargs,
                    atol = 1e2,
                    rtol = 1e0,
                    initialize = True,
                    incomplete = True,
                    post_iter_save_vars = None,
                    post_ini_callback = None,
                    starting_i = 1):
    """
    """
    s = ' ::: performing assimilation process with %i max iterations ::: '
    print_text(s % max_iter, cls=self.this)

    # retain base install directory :
    out_dir_i = self.out_dir

    # directory for saving convergence history :
    d_hist = self.out_dir + 'convergence_history/'
    if not os.path.exists(d_hist) and self.MPI_rank == 0:
        os.makedirs(d_hist)

    # number of digits for saving variables :
    n_i = len(str(max_iter))

    # starting time :
    t0 = time()

    # L2 error norm between iterations :
    abs_error = np.inf
    rel_error = np.inf

    # number of iterations, from a starting point (useful for restarts) :
    if starting_i <= 1:
        counter = 1
    else:
        counter = starting_i

    # initialize friction field :
    self.init_beta(beta_i, cls=self.this)

    # previous friction for norm calculation :
    beta_prev = self.beta.copy(True)

    # perform initialization step if desired :
    if initialize:
        s = ' - performing initialization step - '
        print_text(s, cls=self.this)

    # set the initialization output directory :
    out_dir_n = 'initialization/'
    self.set_out_dir(out_dir_i + out_dir_n)

    # thermo-mechanical couple :
    self.thermo_solve(**tmc_kwargs)

    # call the post function if set :
    if post_ini_callback is not None:
        s = ' ::: calling post-initialization assimilate_U_ob ' + \
            'callback function ::: '
        print_text(s, cls=self.this)
        post_ini_callback()

    # otherwise, tell us that we are not initializing :
    else:
        s = ' - skipping initialization step - '
        print_text(s, cls=self.this)

    # save the v_opt bounds on Fb :
    bounds = copy(tmc_kwargs['wop_kwargs']['bounds'])

    # assimilate the data :
    while abs_error > atol and rel_error > rtol and counter <= max_iter:
        s = ' ::: entering iterate %i of %i of assimilation process ::: '
        print_text(s % (counter, max_iter), cls=self.this)

        # set a new unique output directory :
        out_dir_n = '%0*d/' % (n_i, counter)
        self.set_out_dir(out_dir_i + out_dir_n)

        # the incomplete adjoint means the viscosity is linear, and
        # we do not want to reset the original momentum configuration, because
        # we have more non-linear solves to do :
        if incomplete and not momentum.linear:
            momentum.linearize_viscosity(reset_orig_config=True)

        # re-initialize friction field :
        if counter > starting_i: self.init_beta(beta_i, cls=self.this)

        # optimize the velocity :
        momentum.optimize_U_ob(**uop_kwargs)

        # reset the momentum to the original configuration :
        if not momentum.linear_s and momentum.linear: momentum.reset()

        # thermo-mechanically couple :
        self.thermo_solve(**tmc_kwargs)

        # calculate L2 norms :
        abs_error_n = norm(beta_prev.vector() - self.beta.vector(), 'l2')
        beta_nrm = norm(self.beta.vector(), 'l2')

        # save convergence history :
        if counter == 1:
            rel_error = abs_error_n
            if self.MPI_rank == 0:
                err_a = np.array([abs_error_n])
                nrm_a = np.array([beta_nrm])
                np.savetxt(d_hist + 'abs_err.txt', err_a)
                np.savetxt(d_hist + 'beta_norm.txt', nrm_a)
        else:
            rel_error = abs(abs_error - abs_error_n)
            if self.MPI_rank == 0:
                err_n = np.loadtxt(d_hist + 'abs_err.txt')
                nrm_n = np.loadtxt(d_hist + 'beta_norm.txt')
                err_a = np.append(err_n, np.array([abs_error_n]))
                nrm_a = np.append(nrm_n, np.array([beta_nrm]))
                np.savetxt(d_hist + 'abs_err.txt', err_a)
                np.savetxt(d_hist + 'beta_norm.txt', nrm_a)

        # print info to screen :
        if self.MPI_rank == 0:
            s0 = '>>> '
            s1 = 'U_ob assimilation iteration %i (max %i) done: ' % \
                (counter, max_iter)
            s2 = 'r (abs) = %.2e ' % abs_error
            s3 = ' (tol %.2e) ' % atol
            s4 = 'r (rel) = %.2e ' % rel_error
            s5 = ' (tol %.2e) ' % rtol
            s6 = '<<<'
            text0 = get_text(s0, 'red', 1)
            text1 = get_text(s1, 'red')
            text2 = get_text(s2, 'red', 1)
            text3 = get_text(s3, 'red')

```

```

            text4 = get_text(s4, 'red', 1)
            text5 = get_text(s5, 'red')
            text6 = get_text(s6, 'red', 1)
            print text0 + text1 + text2 + text3 + text4 + text5 + text6

    # save state to unique hdf5 file :
    if isinstance(post_iter_save_vars, list):
        s = ' ::: saving variables in list arg post_iter_save_vars ::: '
        print_text(s, cls=self.this)
        out_file = self.out_dir + 'inverted.h5'
        foutput = HDF5File(mpi_comm_world(), out_file, 'w')
        for var in post_iter_save_vars:
            self.save_hdf5(var, f=foutput)
        foutput.close()

    # update error stuff and increment iteration counter :
    abs_error = abs_error_n
    beta_prev = self.beta.copy(True)
    counter += 1

    # calculate total time to compute
    tf = time()
    s = tf - t0
    m = s / 60.0
    h = m / 60.0
    s = s % 60
    m = m % 60
    text = "time to compute TMC optimized ||u - u_ob||: %02d:%02d:%02d"
    print_text(text % (h, m, s), 'red', 1)

    # plot the convergence history :
    s = ' ::: convergence info saved to \'s\' ::: '
    print_text(s % d_hist, cls=self.this)
    if self.MPI_rank == 0:
        np.savetxt(d_hist + 'time.txt', np.array([tf - t0]))

    err_a = np.loadtxt(d_hist + 'abs_err.txt')
    nrm_a = np.loadtxt(d_hist + 'beta_norm.txt')

    # plot iteration error :
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_ylabel(r'$\|u - u_{ob}\| - \|beta_n\| - \|beta_n\| \Vert$')
    ax.set_xlabel(r'iteration')
    ax.plot(err_a, 'k-', lw=2.0)
    plt.grid()
    plt.savefig(d_hist + 'abs_err.png', dpi=100)
    plt.close(fig)

    # plot theta norm :
    fig = plt.figure()
    ax = fig.add_subplot(111)
    ax.set_ylabel(r'$\|u - u_{ob}\| - \|beta_n\| \Vert$')
    ax.set_xlabel(r'iteration')
    ax.plot(nrm_a, 'k-', lw=2.0)
    plt.grid()
    plt.savefig(d_hist + 'beta_norm.png', dpi=100)
    plt.close(fig)

```

12.3 L-curve analysis

In order to determine the correct values of γ_3 and γ_4 , we use a process referred to as *L-curve analysis* developed by Hansen (1992). The method relies on the fact that for many inverse problems, the shape of the curve resulting from plotting the regularization functional values against the cost functional values for a series of regularization parameters resembles the shape of an 'L'. The 'corner' of this curve approximately corresponds to the point whereby increasing regularization begins to negatively impact the cost functional minimization with negligible improvement in smoothness of the control parameter. One might say that the 'cost' of regularization after this point becomes too high.

To this end, we have created the function `L_curve` contained within the `Model` class (Code Listing 12.4) for automatically performing this calculation for any child class of the `CSLVR Physics` class (Code Listing 12.3). All of the physics calculations presented in §9, §10, §13, §14, and §15 inherit from this class; this is because any physics calculation may have an application as an optimization problem. In regards to momentum objective (12.1), we plot the values of the total cost functional $J = J_1 + J_2$ versus either regularization functional J_3 or J_4 .

In §12.5 we examine this procedure for a similar problem as that presented in §9.7. Finally, this method was also employed to derive the regularization parameters utilized to generate the simulations presented in Chapter 16.

Code Listing 12.3: CSLVR source code for the abstract class Physics from which all physics calculation classes inherit.

```
from helper import raiseNotDefined
from fenics import *
from dolfin_adjoint import *
from calvr.io import print_text

class Physics(object):
    """
    This abstract class outlines the structure of a physics calculation.
    """
    def __new__(self, model, *args, **kwargs):
        """
        Creates and returns a new Physics object.
        """
        instance = object.__new__(self)
        instance.model = model
        return instance

    def color(self):
        """
        return the default color for this class.
        """
        return 'white'

    def default_solve_params(self):
        """
        Returns a set of default solver parameters that yield good performance
        """
        params = {'solver': 'mumps'}
        return params

    def get_solve_params(self):
        """
        Returns the solve parameters.
        """
        return self.default_solve_params()

    def form_reg_ftn(self, c, integral, kind='TV', alpha=1.0,
                    alpha_tik=1e-7, alpha_tv=10):
        """
        Formulates, and returns the regularization functional for use
        with adjoint, saved to self.R.
        """
        self.alpha = alpha # need to save this for printing values.
        model = self.model
        self.reg_ftn_type = kind # need to save this for printing values.

        # differentiate between regularization over cells or facets :
        if integral in [model.OMEGA_GND, model.OMEGA_FLT]:
            dR = model.dx(integral)
        else:
            dR = model.ds(integral)

        kinds = ['TV', 'Tikhonov', 'TV_Tik_hybrid', 'square', 'abs']

        # form regularization term 'R' :
        if kind not in kinds:
            s = ">>> VALID REGULARIZATIONS ARE 'TV', 'Tikhonov', 'square', ' ' + \
                'abs', or 'TV_Tik_hybrid' <<<"
            print_text(s, 'red', 1)
            sys.exit(1)
        elif kind == 'TV':
            R = alpha * sqrt(inner(grad(c), grad(c)) + 1e-15) * dR
            Rp = sqrt(inner(grad(c), grad(c)) + 1e-15) * dR
            s = "": forming 'TV' regularization functional with parameter" + \
                " alpha = %.2E :": " % alpha
        elif kind == 'Tikhonov':
            R = alpha * 0.5 * inner(grad(c), grad(c)) * dR
            Rp = 0.5 * inner(grad(c), grad(c)) * dR
            s = "": forming 'Tikhonov' regularization functional with parameter" + \
                " alpha = %.2E :": " % alpha
        elif kind == 'TV_Tik_hybrid':
            self.R1 = alpha_tik * 0.5 * inner(grad(c), grad(c)) * dR
            self.R2 = alpha_tv * sqrt(inner(grad(c), grad(c)) + 1e-15) * dR
            R1p = 0.5 * inner(grad(c), grad(c)) * dR
            R2p = sqrt(inner(grad(c), grad(c)) + 1e-15) * dR
            R = self.R1 + self.R2
            Rp = R1p + R2p
            s = "": forming Tikhonov/TV hybrid regularization with alpha_tik = " % \
                "%.1e and alpha_tv = %.1e :": " (alpha_tik, alpha_tv)
        elif kind == 'square':
            R = alpha * 0.5 * c**2 * dR
            Rp = 0.5 * c**2 * dR
            s = "": forming 'square' regularization functional with parameter" + \
                " alpha = %.2E :": " % alpha
        elif kind == 'abs':
            R = alpha * abs(c) * dR
            Rp = abs(c) * dR
            s = "": forming 'abs' regularization functional with parameter" + \
                " alpha = %.2E :": " % alpha
        print_text(s, self.color())
        s = " - integrated over %s -" % model.boundaries[integral]
        print_text(s, self.color())
        self.R = R
        self.Rp = Rp # needed for L-curve

    def solve(self):
        """
        Solves the physics calculation.
        """
        raiseNotDefined()
```

Code Listing 12.4: CSLVR implementation of the L-curve procedure of §12.3. Both the cost and regularization functional values are saved as txt files; and plots of the convergence behavior and L-curve as pdf files.

```
def L_curve(self, alphas, physics, control, int_domain, adj_ftn, adj_kwargs,
            reg_kind='Tikhonov', pre_callback=None, post_callback=None,
            itr_save_vars=None):
    """
    """
    s = "": starting L-curve procedure :":
    print_text(s, cls=self.this)

    # starting time :
    t0 = time()

    # retain base install directory :
    out_dir_i = self.out_dir
```

```
# retain initial control parameter for consistency :
control_ini = control.copy(True)

# functional lists to be populated :
Js = []
Rs = []

# iterate through each of the regularization parameters provided :
for i, alpha in enumerate(alphas):
    s = "": performing L-curve iteration %i with alpha = %.3e :":
    print_text(s % (i, alpha), atrb=1, cls=self.this)

    # reset everything after the first iteration :
    if i > 0:
        s = "": initializing physics :":
        print_text(s, cls=self.this)
        physics.reset()
        self.assign_variable(control, control_ini, cls=self.this)

    # set the appropriate output directory :
    out_dir_n = 'alpha.%.1E/' % alpha
    self.set_out_dir(out_dir_i + out_dir_n)

    # call the pre-adjoint callback function :
    if pre_callback is not None:
        s = "": calling L_curve() pre-adjoint pre_callback() :":
        print_text(s, cls=self.this)
        pre_callback()

    # get new regularization functional :
    R = physics.form_reg_ftn(control, integral=int_domain,
                            kind=reg_kind, alpha=alpha)

    # solve the adjoint system :
    adj_ftn(**adj_kwargs)

    # calculate functionals of interest :
    Rs.append(assemble(physics.Rp))
    Js.append(assemble(physics.Jp))

    # call the pre-adjoint callback function :
    if post_callback is not None:
        s = "": calling L_curve() post-adjoint post_callback() :":
        print_text(s, cls=self.this)
        post_callback()

    # save state to unique hdf5 file :
    if isinstance(itr_save_vars, list):
        s = "": saving variables in list arg itr_save_vars :":
        print_text(s, cls=self.this)
        out_file = self.out_dir + 'lcurve.h5'
        foutput = HDF5File(mpi_comm_world(), out_file, 'w')
        for var in itr_save_vars:
            self.save_hdf5(var, f=foutput)
        foutput.close()

s = "": L-curve procedure complete :":
print_text(s, cls=self.this)

# calculate total time to compute
s = time() - t0
m = s / 60.0
h = m / 60.0
s = s % 60
m = m % 60
text = "time to complete L-curve procedure: %02d:%02d:%02d" % (h, m, s)
print_text(text, 'red', 1)

# save the resulting functional values and alphas to CSF :
if self.MPI_rank==0:
    # iterate through the directories we just created and grab the data :
    alphas = []
    Js = []
    Js = []
    J1s = []
    J2s = []
    Rs = []
    ns = []
    for d in next(os.walk(out_dir_i))[1]:
        m = re.search('(alpha_)(\d+W\dE\dW\d+)', d)
        if m is not None:
            do = out_dir_i + d + '/objective_ftnls_history/'
            alphas.append(float(m.group(2)))
            Js.append(np.loadtxt(do + 'Ds.txt'))
            Js.append(np.loadtxt(do + 'Js.txt'))
            J1s.append(np.loadtxt(do + 'J1s.txt'))
            J2s.append(np.loadtxt(do + 'J2s.txt'))
            Rs.append(np.loadtxt(do + 'Rs.txt'))
            ns.append(len(Js[-1]))
    alphas = np.array(alphas)
    Js = np.array(Js)
    Js = np.array(Js)
    J1s = np.array(J1s)
    J2s = np.array(J2s)
    Rs = np.array(Rs)
    ns = np.array(ns)

    # sort everything :
    idx = np.argsort(alphas)
    alphas = alphas[idx]
    Js = Js[idx]
    Js = Js[idx]
    J1s = J1s[idx]
    J2s = J2s[idx]
    Rs = Rs[idx]
    ns = ns[idx]

    # plot the functionals :
    #=====
    fig = plt.figure(figsize=(6,2.5))
    ax = fig.add_subplot(111)

    # we want to plot the different alpha values a different shade :
    cmap = plt.get_cmap('viridis')
    colors = [cmap(x) for x in np.linspace(0, 1, len(alphas))]

    k = 0 # counter so we can plot side-by-side
    ints = [0] # to modify the x-axis labels
    for i, c in zip(range(len(alphas)), colors):
        xi = np.arange(k, k + ns[i])
        ints.append(xi.max())
        # if this is the first iteration, we put a legend on it :
        if i == 0:
            # if we have two cost functionals, let's plot both :
            if physics.obj_ftn_type == 'log_L2_hybrid':
                ax.plot(xi, J1s[i], '-', c='0.5', lw=2.0,
                        label = r'$\mathscr{J}_1$')
                ax.plot(xi, J2s[i], '-', c='k', lw=2.0,
                        label = r'$\mathscr{J}_2$')
            # otherwise, just the one :
            else:
                ax.plot(xi, Js[i], '-', c='k', lw=2.0,
                        label = r'$\mathscr{J}$')
        # always plot the regularization functional :
        ax.plot(xi, Rs[i], '-', c='r', lw=2.0,
```

```

        label = r'\mathscr{R}'
    # otherwise, we don't need cluttered legends :
    else:
        # if we have two cost functionals, let's plot both :
        if physics.obj_ftn_type == 'log_L2_hybrid':
            ax.plot(xi, Js[i], '-', c='k', lw=2.0)
            ax.plot(xi, J2s[i], '-', c='k', lw=2.0)
        # otherwise, just the one :
        else:
            ax.plot(xi, Js[i], '-', c='k', lw=2.0)
        # always plot the regularization functional :
        ax.plot(xi, Rs[i], '-', c='r', lw=2.0)
        k += ns[i] - 1
    ints = np.array(ints)

    label = []
    for i in alphas:
        label.append(r'\gamma = %g' % i)

    # reset the x-label to be meaningful :
    ax.set_xticks(ints)
    ax.set_xticklabels(label, size='small', ha='left', rotation=-45)
    ax.set_xlabel(r'relative iteration')

    ax.grid()
    ax.set_yscale('log')

    # plot the functional legend across the top in a row :
    if physics.obj_ftn_type == 'log_L2_hybrid': ncol = 3
    else: ncol = 2
    leg = ax.legend(loc='upper center', ncol=ncol)
    leg.get_frame().set_alpha(0.0)

    plt.tight_layout()
    plt.savefig(out_dir_i + 'convergence.pdf')
    plt.close(fig)

    # plot L-curve :
    #=====
    # we only want the last value of each optimization :
    fin Js = Js[-1]
    fin Rs = Rs[-1]

    fig = plt.figure(figsize=(6,2.5))
    ax = fig.add_subplot(111)

    ax.plot(fin Js, fin Rs, 'k-', lw=2.0)

    ax.grid()

    # useful for figuring out what reg. parameter goes with what :
    for i,c in zip(range(len(alphas)), colors):
        ax.plot(fin Js[i], fin Rs[i], 'o', c=c, lw=2.0,
            label = r'\gamma = %g' % alphas[i])

    ax.set_xlabel(r'\mathscr{I}~')
    ax.set_ylabel(r'\mathscr{R}~')

    leg = ax.legend(loc='upper right', ncol=2)
    leg.get_frame().set_alpha(0.0)

    ax.set_yscale('log')
    #ax.set_xscale('log')

    plt.tight_layout()
    plt.savefig(out_dir_i + 'l_curve.pdf')
    plt.close(fig)

    # save the functionals :
    #=====
    d = out_dir_i + 'functionals/'
    if not os.path.exists(d):
        os.makedirs(d)
    np.savetxt(d + 'Rs.txt', np.array(fin Rs))
    np.savetxt(d + 'Js.txt', np.array(fin Js))
    np.savetxt(d + 'as.txt', np.array(alphas))

```

12.4 Ice-shelf inversion procedure

Due to the fact that basal traction β defined over floating ice-shelves is very close to zero (Greve and Blatter, 2009), a different control parameter must be specified in order to match the surface velocity observations in these areas. One choice for this control is enhancement factor E in flow-rate factor (10.22). The inversion for this parameter has been used to generate preliminary continent-scale inversions of Antarctica with low error velocity misfit $\|\mathbf{u} - \mathbf{u}_{ob}\|$ over ice-shelves. We only make note of the fact that this option is easily implemented with CSLVR, and results in a depth-varying distribution of enhancement E .

12.5 ISMIP-HOM inverse test simulation

For a simple test of the momentum optimization procedure described in §12.1, an inverse form of the ISMIP-HOM project presented previously in §9.7 is performed (Pattyn et al., 2008).

This test is defined over the domain $\Omega \in [0, \ell] \times [0, \ell] \times [B, S] \subset \mathbb{R}^3$ with $k_x \times k_y \times k_z$ node discretization, and specifies the use of a surface height with uniform slope $\|\nabla S\| = a$

$$S(x) = -x \tan(a)$$

and basal topography matching the surface slope

$$B(x, y) = S(x) - H,$$

with ice thickness H . As before, we enforce continuity via the periodic \mathbf{u} boundary conditions (the first-order momentum model does not solve for pressure p)

$$\mathbf{u}(0, 0) = \mathbf{u}(\ell, \ell)$$

$$\mathbf{u}(0, \ell) = \mathbf{u}(\ell, 0)$$

$$\mathbf{u}(x, 0) = \mathbf{u}(x, \ell)$$

$$\mathbf{u}(0, y) = \mathbf{u}(\ell, y).$$

To begin, first-order momentum system (9.32) is solved using the ‘true’ basal traction field

$$\beta_{\text{true}}(x, y) = \left(\frac{\beta_{\max}}{2} \right) \sin\left(\frac{2\pi}{\ell} x\right) \sin\left(\frac{2\pi}{\ell} y\right) + \left(\frac{\beta_{\max}}{2} \right)$$

with maximum value β_{\max} (Figure 12.5). Similar to Petra et al. (2012), we add normally-distributed-random noise to the resulting ‘true’ velocity field \mathbf{u}_{true} with standard deviation σ to create the simulated ‘observed’ velocity

$$\mathbf{u}_{ob} = \mathbf{u}_{\text{true}} + \epsilon$$

where

$$\epsilon \stackrel{iid}{\sim} \mathcal{N}(\mathbf{0}, \sigma^2 I), \quad \sigma = \frac{\|\mathbf{u}_{\text{true}}\|_{\infty}}{\text{SNR}}$$

with signal-to-noise ratio SNR (Figure 12.5).

For simplicity, we use the isothermal rate-factor $A = 10^{-16}$ for use with viscosity η , thus removing the necessity to optimize energy θ . Table 12.1 lists the coefficients and values used.

To begin the inversion process, L^2 cost functional coefficient γ_1 and logarithmic cost functional coefficient γ_2 in (12.1) are determined by solving momentum optimization problem (12.7) and adjusting their relative values such that at the end of the optimization their associated functionals are of approximately the same order. Following Morlighem, Seroussi, and Rignot (2013), we set $\gamma_1 = 1$ and derive by this process $\gamma_2 = 10^5$ (Figures 12.1 and 12.3).

The next step is to derive a proper value for the regularization parameters γ_3 and γ_4 associated respectively with Tikhonov regularization functional (12.4) and total variation regularization functional (12.5). To this end, we perform the L-curve procedure described in §12.3 over a range of Tikhonov parameters γ_3 and TV parameters γ_4 in objective (12.1) (see Code Listing 12.5). For simplicity, we vary only one of γ_3 or γ_4 and set the other to zero (Figures 12.2 and 12.4).

Results generated with Code Listing 12.6 indicate that an appropriate value for both parameters is $\gamma_3, \gamma_4 = 100$. However, the traction field resulting from Tikhonov regularization

Table 12.1: ISMIP-HOM inverse variables.

Variable	Value	Units	Description
$\dot{\epsilon}_0$	10^{-15}	a^{-1}	strain regularization
A	10^{-16}	$\text{Pa}^{-3} \text{a}^{-1}$	flow-rate factor
ℓ	20	km	width of domain
a	0.5	o	surface gradient mag.
H	1000	m	ice thickness
β_i	β_{SIA}	$\text{kg m}^{-2} \text{a}^{-1}$	ini. traction coef.
SNR	100	—	\mathbf{u}_{ob} signal-to-noise ratio
γ_1	10^{-2}	$\text{kg m}^{-2} \text{a}^{-1}$	L^2 cost coefficient
γ_2	5×10^3	J a^{-1}	log. cost coefficient
γ_3	10^{-1}	$\text{m}^6 \text{kg}^{-1} \text{a}^{-1}$	Tikhonov reg. coef.
γ_4	10	$\text{m}^6 \text{kg}^{-1} \text{a}^{-1}$	TV reg. coeff.
F_b	0	m a^{-1}	basal water discharge
k_x	15	—	number of x divisions
k_y	15	—	number of y divisions
k_z	5	—	number of z divisions
N_e	6750	—	number of cells
N_n	1536	—	number of vertices

with $\gamma_3 = 100$, $\gamma_4 = 0$ are much more irregular than the results obtained using TV-regularization with $\gamma_3 = 0$, $\gamma_4 = 100$ (compare Figures 12.6 and 12.7). Results obtained using Tikhonov-regularization with $\gamma_3 = 500$, $\gamma_4 = 0$ produced a qualitatively-smoother result, closer to that obtained via TV-regularization with $\gamma_3 = 0$, $\gamma_4 = 100$ (Figure 12.8).

We conclude by noting that the L-curve procedure described in §12.3 is a means to derive values for regularization parameters that are *approximately* to optimal. Thus simulation and examination of results may be required in order to derive an appropriate value for these parameters. Additionally, when the true value of the unknown quantity is known, such as the case here, the regularization parameter may be chosen to minimize the error $\|\mathbf{u}^* - \mathbf{u}_{\text{true}}\|$. Real-world data assimilations such as that presented in Chapter 16 do not include ‘true’ values for the velocity, and so we rely on the technique of trial-and-error to derive these parameters.

Code Listing 12.5: CSLVR script for performing the L-curve procedure.

```
from csivr import *
from scipy import random

#reg_type = 'TV'
reg_type = 'Tikhonov'
out_dir = './L_curve_results/' + reg_type + '/'
plt_dir = './../images/data_assimilation/ISMIP_HOM_C/' + reg_type + '/'

a = 0.5 * pi / 180
L = 20000
bmax = 1000

p1 = Point(0.0, 0.0, 0.0)
p2 = Point(L, L, 1)
mesh = BoxMesh(p1, p2, 15, 15, 5)

model = D3Model(mesh, out_dir = out_dir, use_periodic = True)

surface = Expression('- x[0] * tan(a)', a=a,
                    element=model.Q.ufl_element())
bed = Expression('- x[0] * tan(a) - 1000.0', a=a,
                element=model.Q.ufl_element())
beta = Expression('bmax/2 + bmax/2 * sin(2*pi*x[0]/L) * sin(2*pi*x[1]/L)',
                bmax=bmax, L=L, element=model.Q.ufl_element())

# calculate the boundaries for integration :
model.calculate_boundaries()

# deform the mesh to the desired geometry :
model.deform_mesh_to_geometry(surface, bed)

# initialize important variables :
model.init_beta(beta) # traction
model.init_A(1e-16) # isothermal rate-factor

mom = MomentumDukowiczBP(model)
```

```
mom.solve(annotate=False)

# add noise with a signal-to-noise ratio of 100 :
snr = 100.0
u = Function(model.Q)
v = Function(model.Q)
assign(u, model.U3.sub(0))
assign(v, model.U3.sub(1))
u_o = u.vector().array()
v_o = v.vector().array()
n = len(u_o)
sig = model.get_norm(as_vector([u, v]), 'linf')[1] / snr
print_min_max(snr, 'SNR')
print_min_max(sig, 'sigma')

u_error = sig * random.randn(n)
v_error = sig * random.randn(n)
u_ob = u_o + u_error
v_ob = v_o + v_error

# init the 'observed' velocity :
model.init_U_ob(u_ob, v_ob)
u_ob_ex = model.vert.extrude(model.u_ob, 'down')
v_ob_ex = model.vert.extrude(model.v_ob, 'down')
model.init_U_ob(u_ob_ex, v_ob_ex)

# init the traction to the SIA approximation :
model.init_beta_SIA()

# form the cost functional :
mom.form_obj_ftn(integral=model.GAMMA_U_GND, kind='log_L2_hybrid',
                g1=1, g2=1e5)

# solving the incomplete adjoint is more efficient :
mom.linearize_viscosity()

# optimize for beta :
adj_kwargs = {'control' : model.beta,
              'bounds' : (1e-5, 1e7),
              'method' : 'ipopt',
              'max_iter' : 100,
              'adj_save_vars' : None,
              'adj_callback' : None,
              'post_adj_callback' : None}

# regularization parameters :
alphas = [1e-3, 1e-2, 1e-1, 1e0, 1e1, 1e2, 1e3, 1e4]

model.L_curve(alphas = alphas,
              physics = mom,
              control = model.beta,
              int_domain = model.GAMMA_B_GND,
              adj_ftn = mom.optimize_U_ob,
              adj_kwargs = adj_kwargs,
              reg_kind = reg_type,
              pre_callback = None,
              post_callback = None,
              itr_save_vars = None)
```

Code Listing 12.6: CSLVR script used to solve the inverse ISMIP-HOM experiment with regularization parameters derived by Code Listing 12.5.

```
from csivr import *
from scipy import random

# out directories for saving data and images :
#reg_type = 'TV'
reg_type = 'Tikhonov'
#reg_type = 'TV_Tik_hybrid'
#reg_type = 'ISMIP_HOM_C_inverse' + reg_type + '_results/'
plt_dir = './../images/data_assimilation/ISMIP_HOM_C/' + reg_type + '/'

# constants used :
a = 0.5 * pi / 180
L = 20000
bmax = 1000

# a generic box mesh that will be fit to geometry below :
p1 = Point(0.0, 0.0, 0.0)
p2 = Point(L, L, 1)
mesh = BoxMesh(p1, p2, 15, 15, 5)

# this is a 3D model :
model = D3Model(mesh, out_dir = out_dir + 'true/', use_periodic = True)

# expressions for the surface and basal topography, and friction :
surface = Expression('- x[0] * tan(a)', a=a,
                    element=model.Q.ufl_element())
bed = Expression('- x[0] * tan(a) - 1000.0', a=a,
                element=model.Q.ufl_element())
beta = Expression('bmax/2 + bmax/2 * sin(2*pi*x[0]/L) * sin(2*pi*x[1]/L)',
                bmax=bmax, L=L, element=model.Q.ufl_element())

# calculate the boundaries for integration :
model.calculate_boundaries()

# deform the mesh to the desired geometry :
model.deform_mesh_to_geometry(surface, bed)

# create the bed and surface meshes, for plotting purposes :
model.form_bed_mesh()
model.form_srf_mesh()

# create 2D models, again for plotting only :
bedmodel = D2Model(model.bedmesh, out_dir)
srfmodel = D2Model(model.srfmesh, out_dir)

# initialize important variables :
model.init_beta(beta) # traction
model.init_A(1e-16) # isothermal rate-factor

# create the first-order momentum object and solve :
mom = MomentumDukowiczBP(model)
mom.solve(annotate=False)

# add noise with a signal-to-noise ratio of 100 :
snr = 100.0
u = Function(model.Q)
v = Function(model.Q)
assign(u, model.U3.sub(0))
assign(v, model.U3.sub(1))
u_o = u.vector().array()
v_o = v.vector().array()
n = len(u_o)
sig = model.get_norm(as_vector([u, v]), 'linf')[1] / snr
print_min_max(snr, 'SNR')
print_min_max(sig, 'sigma')
```

```

u_error = sig * random.randn(n)
v_error = sig * random.randn(n)
u_ob = u_o + u_error
v_ob = v_o + v_error

# init the 'observed' velocity :
model.init_U_ob(u_ob, v_ob)
u_ob_ex = model.vert_extrude(model.u_ob, 'down')
v_ob_ex = model.vert_extrude(model.v_ob, 'down')
model.init_U_ob(u_ob_ex, v_ob_ex)

# assign variables to the submesh for plotting :
bedmodel.assign_submesh_variable(bedmodel.beta, model.beta)
srfmodel.assign_submesh_variable(srfmodel.U_ob, model.U_ob)
srfmodel.assign_submesh_variable(srfmodel.u_ob, model.u_ob)
srfmodel.assign_submesh_variable(srfmodel.v_ob, model.v_ob)
srfmodel.assign_submesh_variable(srfmodel.U3, model.U3)
srfmodel.init_U_mag(srfmodel.U3)

# zero out the vertical velocity for comparison :
srfmodel.init_w(Constant(0.0))

# plotting :
beta_min = bedmodel.beta.vector().min()
beta_max = bedmodel.beta.vector().max()
beta_lvls = array([beta_min, 100, 200, 300, 400, 500, 600,
                  700, 800, 900, beta_max])
plot_variable(u = bedmodel.beta, name = 'beta_true', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = beta_lvls, tp = True,
              show = False, cb_format='%i', hide_ax_tick_labels=True)

U_min = srfmodel.U_mag.vector().min()
U_max = srfmodel.U_mag.vector().max()
U_lvls = array([U_min, 170, 180, 200, 220, 240, 260, 280, U_max])
plot_variable(u = srfmodel.U3, name = 'U_true', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = U_lvls, tp = True,
              show = False, cb_format='%i', hide_ax_tick_labels=True)

U_ob_min = srfmodel.U_ob.vector().min()
U_ob_max = srfmodel.U_ob.vector().max()
U_ob_lvls = array([U_ob_min, 170, 180, 200, 220, 240, 260, 280, U_ob_max])
U_ob = as_vector([srfmodel.u_ob, srfmodel.v_ob])
plot_variable(u = U_ob, name = 'U_ob', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = U_ob_lvls, tp = True,
              show = False, cb_format='%i', hide_ax_tick_labels=True)

# calculate the initial tracin field from the SIA approximation :
model.init_beta_SIA()

# model.beta has been reassigned, so let's plot it :
bedmodel.assign_submesh_variable(bedmodel.beta, model.beta)
beta_min = bedmodel.beta.vector().min()
beta_max = bedmodel.beta.vector().max()
beta_lvls = array([250, 275, 300, 325, 350, 375, 400, 425, 450, beta_max])
plot_variable(u = bedmodel.beta, name = 'beta_SIA', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = beta_lvls, tp = True,
              show = False, cb_format='%i', hide_ax_tick_labels=True,
              extend = 'min')

model.set_out_dir(out_dir + 'inversion/')

# post-adjoint-iteration callback function :
def adj_post_cb_ftn():
    """
    this is called when the optimization is done. Here all we do is plot, but
    you may want to calculate other variables of interest to :
    """
    bedmodel.assign_submesh_variable(bedmodel.beta, model.beta)
    srfmodel.assign_submesh_variable(srfmodel.U3, model.U3)
    srfmodel.init_U_mag(srfmodel.U3)
    srfmodel.init_w(Constant(0.0))

# plot beta optimal :
beta_min = bedmodel.beta.vector().min()
beta_max = bedmodel.beta.vector().max()
beta_lvls = array([beta_min, 100, 200, 300, 400, 500, 600, 700,
                  800, 900, beta_max])
plot_variable(u = bedmodel.beta, name = 'beta_opt', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = beta_lvls,
              tp = True, show = False, cb_format='%i',
              hide_ax_tick_labels=True)

# plot u optimal :
U_min = srfmodel.U_mag.vector().min()
U_max = srfmodel.U_mag.vector().max()
U_lvls = array([U_min, 170, 180, 200, 220, 240, 260, 280, U_max])
plot_variable(u = srfmodel.U3, name = 'U_opt', direc = plt_dir,
              cmap = 'viridis', figsize = (6,5), levels = U_lvls, tp = True,
              show = False, cb_format='%i', hide_ax_tick_labels=True)

# or we could save the 3D optimized velocity and beta fields for
# viewing with paraview, like this :
model.save_xdmf(model.U3, 'U_opt')
model.save_xdmf(model.beta, 'beta_opt')

# after every completed adjoining, save the state of these functions :
adj_save_vars = [model.beta, model.U3]

# form the cost functional :
mom.form_obj_ftn(integral=model.GAMMA_U_GND, kind='log_L2_hybrid',
                g1=1, g2=1e5)

# form the regularization functional :
if reg_type == 'TV':
    mom.form_reg_ftn(model.beta, integral=model.GAMMA_B_GND, kind='TV',
                    alpha=100.0)
elif reg_type == 'Tikhonov':
    mom.form_reg_ftn(model.beta, integral=model.GAMMA_B_GND, kind='Tikhonov',
                    alpha=500.0)
elif reg_type == 'TV_Tik_hybrid':
    mom.form_reg_ftn(model.beta, integral=model.GAMMA_B_GND,
                    kind='TV_Tik_hybrid', alpha_tik=250, alpha_tv=50)

# solving the incomplete adjoint is more efficient :
mom.linearize_viscosity()

# optimize for beta :
mom.optimize_U_ob(control = model.beta,
                  bounds = (1e-5, 1e7),
                  method = 'ipopt',
                  max_iter = 100,
                  adj_save_vars = adj_save_vars,
                  adj_callback = None,
                  post_adj_callback = adj_post_cb_ftn)

```

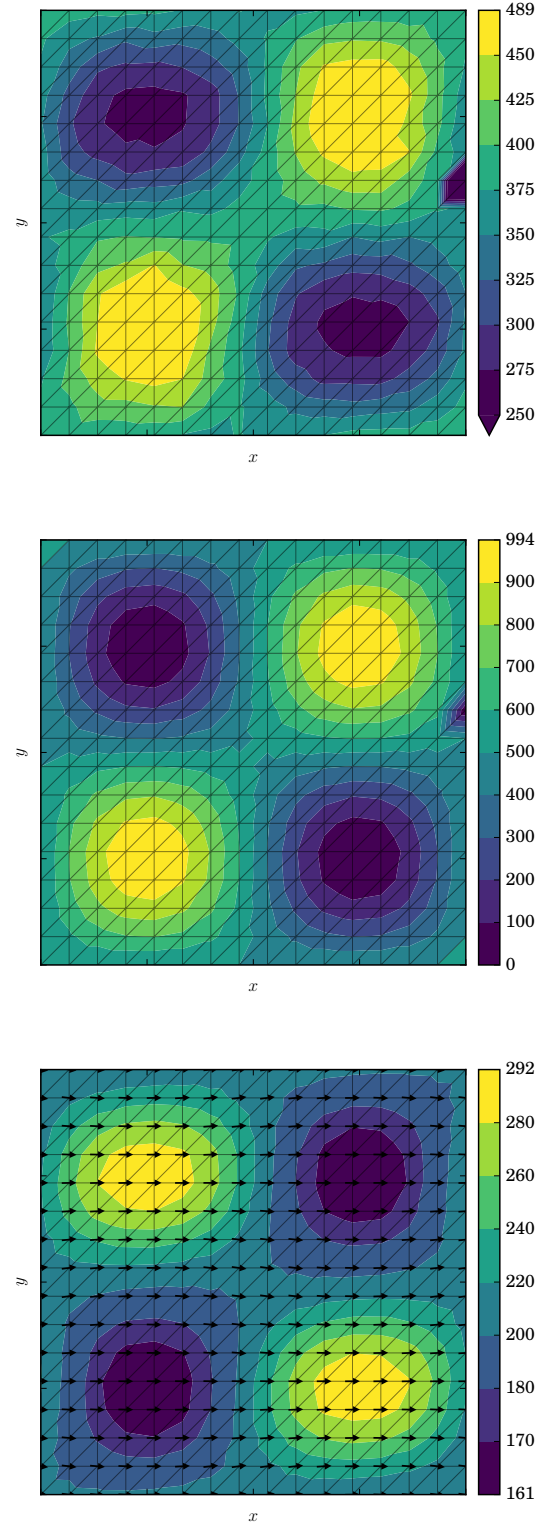


Figure 12.5: ISMIP-HOM C initial value for traction β , SIA traction coefficient β_{SIA} (top), 'true' traction coefficient β_{true} (middle), and 'true' velocity \mathbf{u}_{true} (bottom) with a 20×20 square km grid.

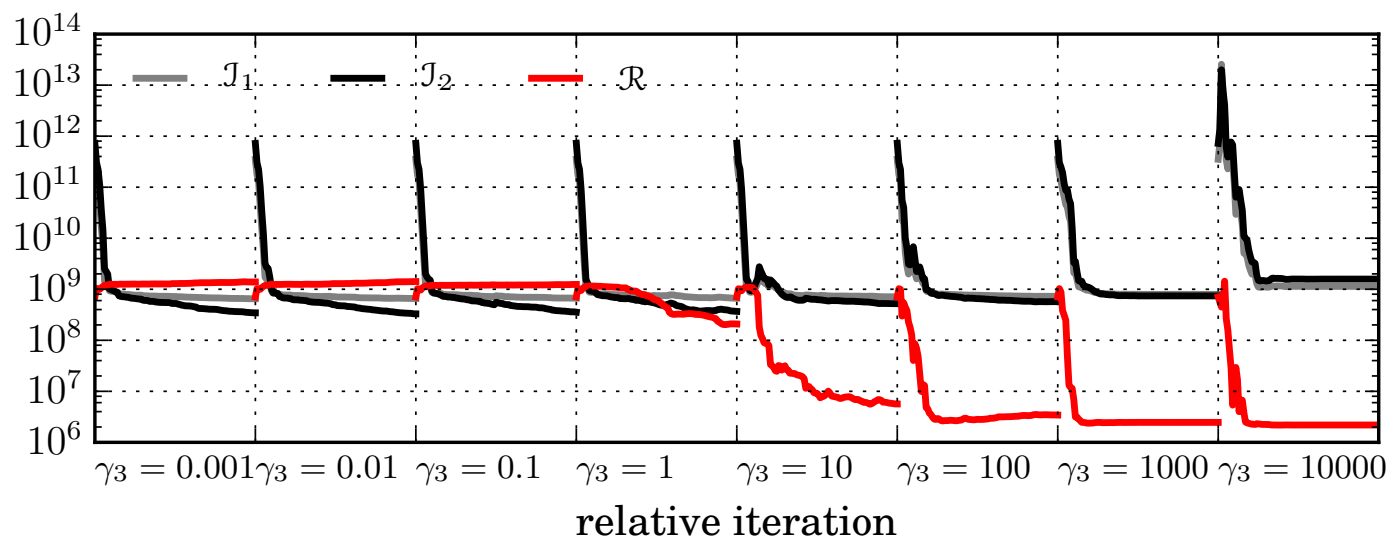


Figure 12.1: Convergence plot of the cost functional $\gamma_1 \mathcal{J}_1$ (grey) and $\gamma_2 \mathcal{J}_2$ (black), and the Tikhonov regularization functional \mathcal{J}_3 (red) for each of the Tikhonov-regularization parameters γ_3 shown on the x-axis. For this procedure, the TV-regularization parameter $\gamma_4 = 0$, and cost functional coefficient chosen to be $\gamma_1 = 1$, $\gamma_2 = 10^5$.

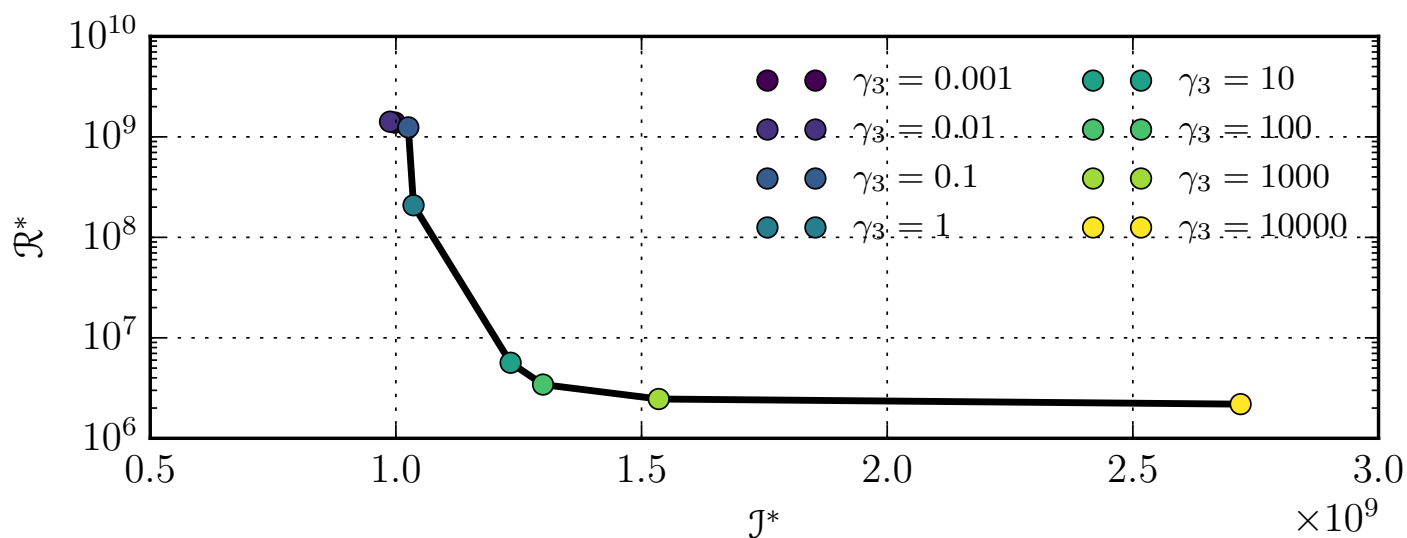


Figure 12.2: L-curve for Tikhonov parameter γ_3 with $\gamma_4 = 0$.

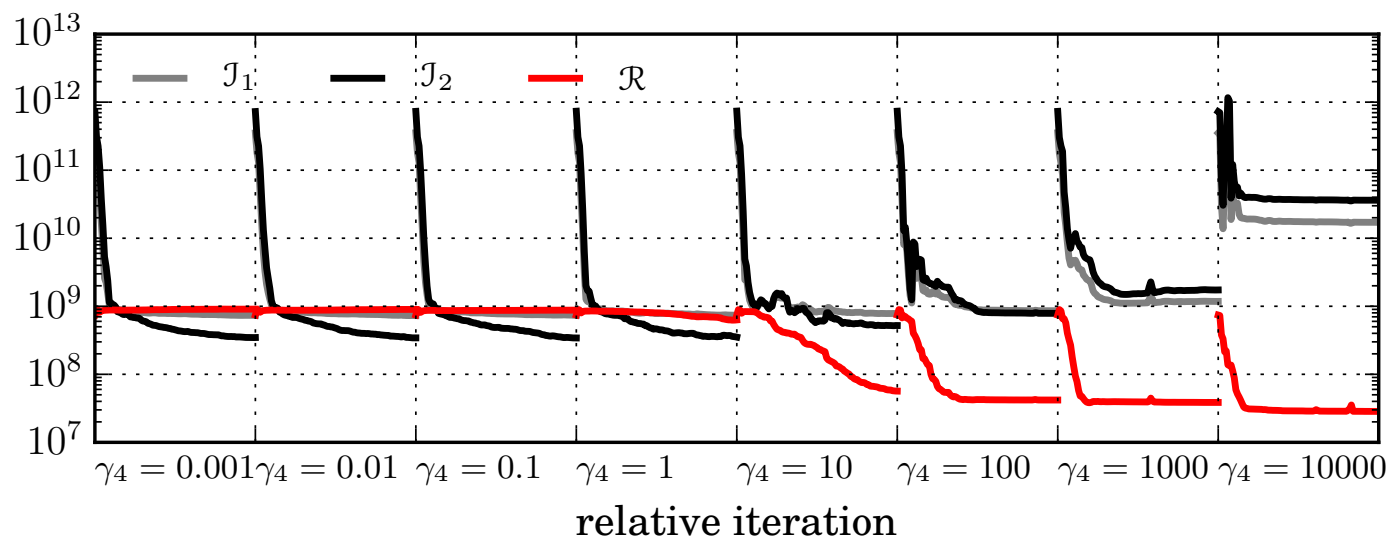


Figure 12.3: Convergence plot of the cost functional $\gamma_1 J_1$ (grey) and $\gamma_2 J_2$ (black), and the TV regularization functional J_4 (red) for each of the TV-regularization parameters γ_4 shown. For this procedure, the Tikhonov-regularization parameter $\gamma_3 = 0$, and cost functional coefficient chosen to be $\gamma_1 = 1$, $\gamma_2 = 10^5$.

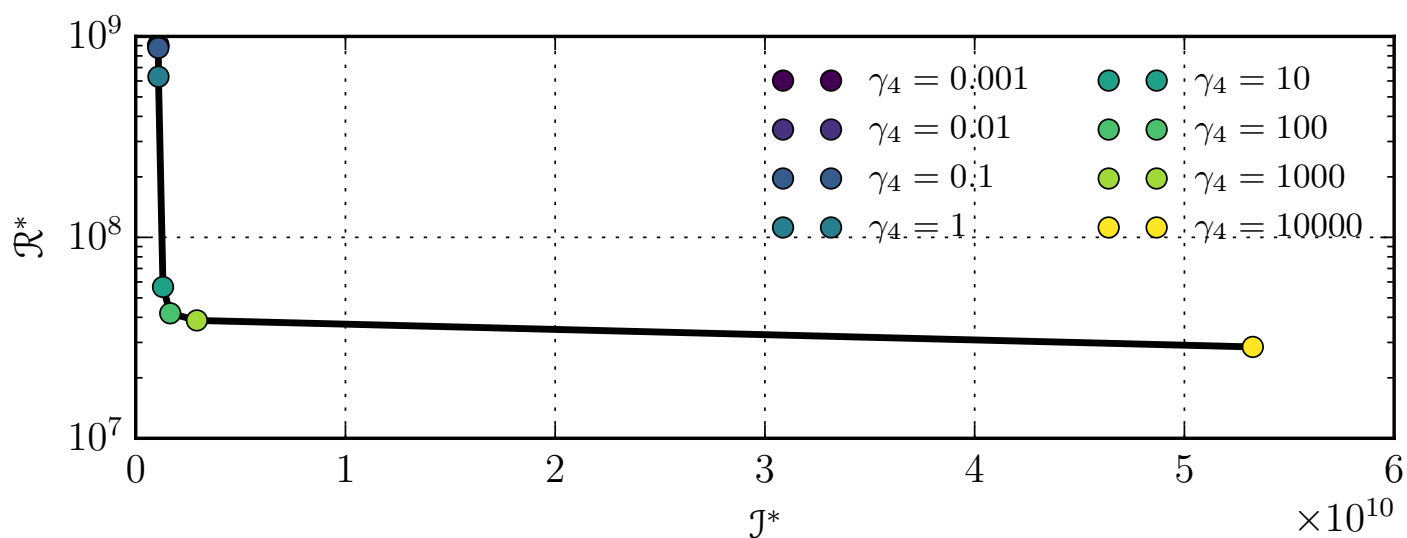


Figure 12.4: L-curve for total-variation parameter γ_4 with $\gamma_3 = 0$.

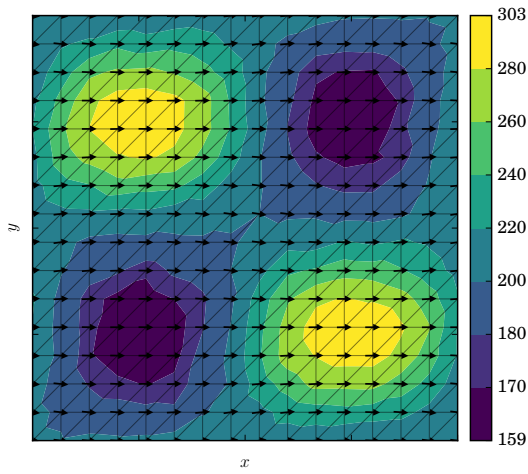
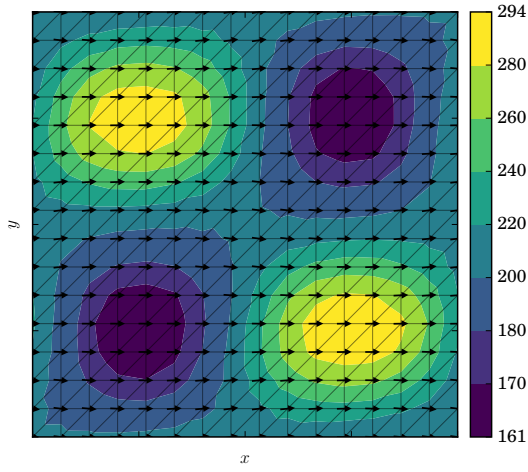
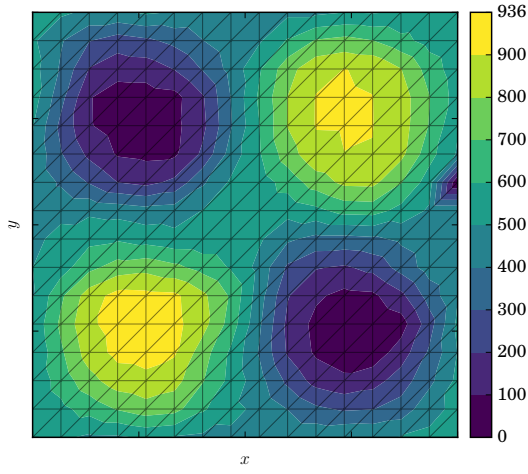


Figure 12.6: Results obtained using TV-regularization with $\gamma_3 = 0$, $\gamma_4 = 100$; optimized traction coefficient β^* (top), optimized velocity \mathbf{u}^* (middle), and ‘observed’ velocity \mathbf{u}_{ob} (bottom) with a 20×20 square km grid.

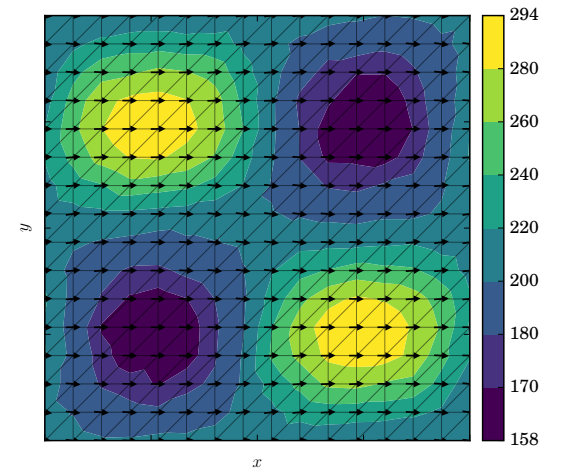
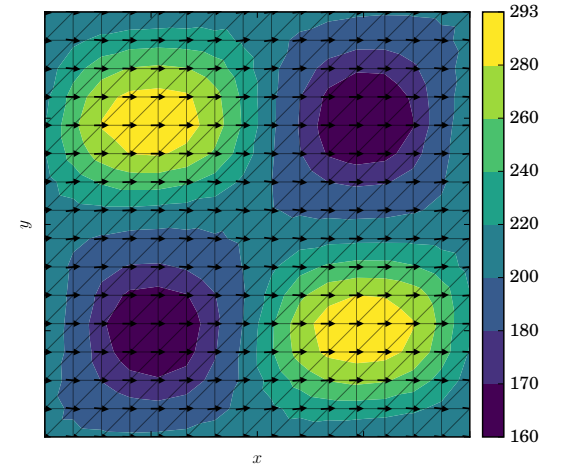
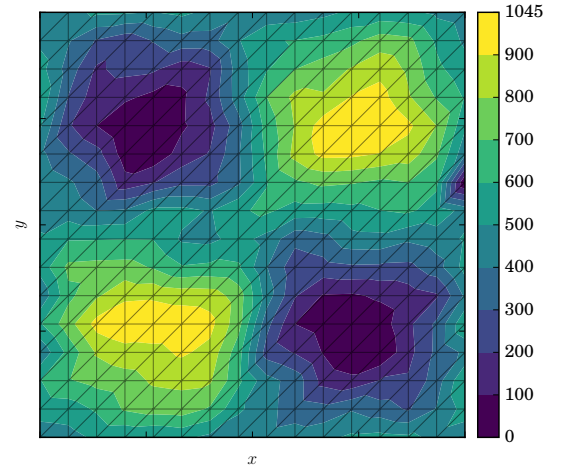


Figure 12.7: Results obtained using Tikhonov-regularization with $\gamma_3 = 100$, $\gamma_4 = 0$; optimized traction coefficient β^* (top), optimized velocity \mathbf{u}^* (middle), and ‘observed’ velocity \mathbf{u}_{ob} (bottom) with a 20×20 square km grid.

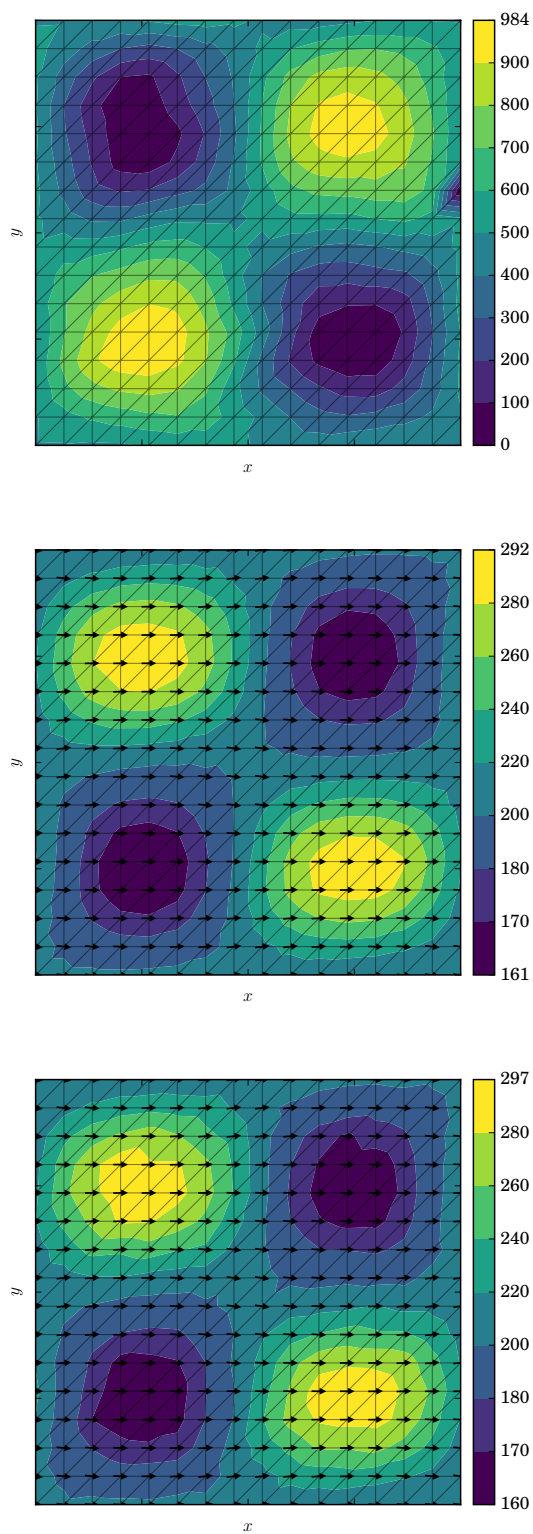


Figure 12.8: Results obtained using Tikhonov-regularization with $\gamma_3 = 500$, $\gamma_4 = 0$; optimized traction coefficient β^* (top), optimized velocity \mathbf{u}^* (middle), and ‘observed’ velocity \mathbf{u}_{ob} (bottom) with a 20×20 square km grid.

Chapter 13

Velocity balance

Due to the complexity of solving the higher-order Stokes models of Chapter 9, it is desirable to be able to run CPU-time-inexpensive computations over continent-scale regions. One way to calculate a less-expensive momentum property is by solving the *balance velocity*; the vertically-averaged velocity

$$\bar{\mathbf{u}} = \frac{1}{H} \int_z \mathbf{u} \, dz, \quad (13.1)$$

with components $\bar{\mathbf{u}} = [\bar{u} \ \bar{v} \ \bar{w}]^\top$.

First, assuming that a given mass within an arbitrary volume of ice $\Omega \in \mathbb{R}^3$ with boundary Γ remains constant, *Leibniz's rule in three dimensions* – better known in continuum mechanics as *Reynold's Transport Theorem* – states that

$$\frac{d}{dt} \int_{\Omega} \rho d\Omega = \int_{\Omega} \frac{\partial \rho}{\partial t} d\Omega + \int_{\Gamma} \rho \mathbf{u} \cdot \mathbf{n} d\Gamma = 0. \quad (13.2)$$

In the context of ice-sheets, there is a flux of ice across the upper surface Γ_S by either accumulation or ablation \dot{a} , and a flux of ice lost across the basal surface Γ_B by melting F_b . Hence

$$(\mathbf{w}_S - \mathbf{u}) \cdot \mathbf{n}|_{\Gamma_S} = \dot{a} \quad (13.3)$$

$$(\mathbf{u} - \mathbf{w}_B) \cdot \mathbf{n}|_{\Gamma_B} = F_b, \quad (13.4)$$

where \mathbf{w}_B and \mathbf{w}_S is the velocity of the free surfaces $F_S(\mathbf{x}, t) = z - S(x, y, t)$ and $F_B(\mathbf{x}, t) = B(x, y, t) - z$ at the coordinate $\mathbf{x} = [x \ y \ z]^\top$, respectively (Greve, 1997). Therefore, the boundary integral of general mass balance relation (13.2) may be decomposed into

$$\begin{aligned} \int_{\Gamma} \rho \mathbf{u} \cdot \mathbf{n} d\Gamma = & \int_{\Gamma_B} \rho (\mathbf{w}_B \cdot \mathbf{n} + F_b) d\Gamma_B + \int_{\Gamma_S} \rho (\mathbf{w}_S \cdot \mathbf{n} - \dot{a}) d\Gamma_S \\ & + \int_{\Gamma_L} \rho \mathbf{u} \cdot \mathbf{n} d\Gamma_L \end{aligned}$$

where Γ_B is the basal surface, Γ_S is the upper surface, and Γ_L is the lateral surface of the volume.

Next, if the rate of change of the free surface $F_S(\mathbf{x}, t)$ and $F_B(\mathbf{x}, t)$ is unchanging, the Eulerian coordinate system demands that

$$\frac{dF_S}{dt} = \frac{\partial F_S}{\partial t} + \mathbf{w}_S \cdot \nabla F_S = 0 \quad (13.5)$$

$$\frac{dF_B}{dt} = \frac{\partial F_B}{\partial t} + \mathbf{w}_B \cdot \nabla F_B = 0. \quad (13.6)$$

Using the outward-pointing normal vector definition on the surface and bed

$$\mathbf{n}|_{\Gamma_S} = \frac{\nabla F_S}{\|\nabla F_S\|}, \quad \mathbf{n}|_{\Gamma_B} = \frac{\nabla F_B}{\|\nabla F_B\|}, \quad (13.7)$$

and relations (13.3, 13.4),

$$\begin{aligned} \mathbf{w}_S \cdot \nabla F_S &= \|\nabla F_S\| \left(\dot{a} + \mathbf{u} \cdot \frac{\nabla F_S}{\|\nabla F_S\|} \right) \\ \mathbf{w}_B \cdot \nabla F_B &= \|\nabla F_B\| \left(-F_b + \mathbf{u} \cdot \frac{\nabla F_B}{\|\nabla F_B\|} \right), \end{aligned}$$

which used in (13.5, 13.6) results in

$$\begin{aligned} \frac{\partial F_S}{\partial t} + \mathbf{u} \cdot \nabla F_S &= -\|\nabla F_S\| \dot{a} \\ \frac{\partial F_B}{\partial t} + \mathbf{u} \cdot \nabla F_B &= \|\nabla F_B\| F_b. \end{aligned}$$

Evaluating the derivatives above,

$$\begin{aligned} \mathbf{u} \cdot \nabla F_S &= \mathbf{u} \cdot (\hat{\mathbf{k}} - \nabla S) = -\|\hat{\mathbf{k}} - \nabla S\| \dot{a} + \frac{\partial S}{\partial t} \\ \mathbf{u} \cdot \nabla F_B &= \mathbf{u} \cdot (\nabla B - \hat{\mathbf{k}}) = \|\nabla B - \hat{\mathbf{k}}\| F_b - \frac{\partial B}{\partial t}, \end{aligned}$$

and with the use of $\hat{\mathbf{k}} \cdot \mathbf{u}(\mathbf{x}, t) = w(\mathbf{x}, t)$,

$$\mathbf{u}|_{\Gamma_S} \cdot \nabla S = w(S) + \|\hat{\mathbf{k}} - \nabla S\| \dot{a} - \frac{\partial S}{\partial t} \quad (13.8)$$

$$\mathbf{u}|_{\Gamma_B} \cdot \nabla B = w(B) + \|\nabla B - \hat{\mathbf{k}}\| F_b - \frac{\partial B}{\partial t}. \quad (13.9)$$

Next, employing the *Divergence Theorem*

$$\int_{\Gamma} \mathbf{v} \cdot \mathbf{n} d\Gamma = \int_{\Omega} \nabla \cdot \mathbf{v} d\Omega \quad (13.10)$$

to the boundary integral in general mass balance (13.2) results in

$$\frac{d}{dt} \int_{\Omega} \rho d\Omega + \int_{\Omega} \rho \nabla \cdot \mathbf{u} d\Omega = \int_{\Omega} \left(\frac{\partial \rho}{\partial t} + \rho \nabla \cdot \mathbf{u} \right) d\Omega = 0,$$

and thus

$$\frac{\partial \rho}{\partial t} + \rho \nabla \cdot \mathbf{u} = 0.$$

Due to the fact that ice is incompressible, $\partial_t \rho = 0$ and conservation of mass relation (8.2), $\nabla \cdot \mathbf{u} = 0$, has been derived. Integrating this expression vertically and applying *Leibniz's Rule* (Appendix B),

$$\int_B^S \nabla \cdot \mathbf{u} dz = \nabla \cdot \left(\int_B^S \mathbf{u} dz \right) + \mathbf{u}|_{\Gamma_B} \cdot \nabla B - \mathbf{u}|_{\Gamma_S} \cdot \nabla S = 0, \quad (13.11)$$

which using balance-velocity definition (13.1) and inserting (13.8, 13.9) provide

$$\begin{aligned} \nabla \cdot (H\bar{\mathbf{u}}) &= +\mathbf{u}|_{\Gamma_S} \cdot \nabla S - \mathbf{u}|_{\Gamma_B} \cdot \nabla B \\ &= + \left(w(S) + \|\hat{\mathbf{k}} - \nabla S\| \dot{a} - \frac{\partial S}{\partial t} \right) \\ &\quad - \left(w(B) + \|\nabla B - \hat{\mathbf{k}}\| F_b - \frac{\partial B}{\partial t} \right) \\ &= \|\hat{\mathbf{k}} - \nabla S\| \dot{a} - \|\nabla B - \hat{\mathbf{k}}\| F_b - \frac{\partial H}{\partial t} + w(S) - w(B) \end{aligned}$$

where in the last step, ice thickness definition $H(x, y) = S(x, y) - B(x, y)$ has been used.

Finally, we can state the *balance velocity equation*

$$\nabla \cdot (H\bar{\mathbf{u}}) = f, \quad (13.12)$$

with forcing term

$$f = \|\hat{\mathbf{k}} - \nabla S\| \dot{a} - \|\nabla B - \hat{\mathbf{k}}\| F_b - \frac{\partial H}{\partial t} + w(S) - w(B). \quad (13.13)$$

Note that $w(S) - w(B)$ is the difference between the vertical component of velocity at the surface to that on the bed, and may be set to values obtained from one of the higher-order models of Chapter 9. The rate of change of the thickness $\partial_t H$ and the surface accumulation/ablation rate \dot{a} may be estimated from observations. Lastly, the mass loss due to basal water melting F_b can be attained by the methods of Chapter 10.

13.1 The direction of flowing ice

Balance velocity $\bar{\mathbf{u}}$ may be decomposed into a direction $\hat{\mathbf{u}}$ and magnitude \bar{u} (Brinkerhoff and Johnson, 2015) such that

$$\bar{\mathbf{u}} = \bar{u} \hat{\mathbf{u}}, \quad (13.14)$$

with unit vector $\hat{\mathbf{u}} = [\hat{u} \ \hat{v} \ \hat{w}]^\top$. From balance equation (13.12),

$$\nabla \cdot (H\bar{u}\hat{\mathbf{u}}) = f, \quad (13.15)$$

which can be expanded with terms H and \hat{u} grouped together,

$$\begin{aligned} \bar{u} H \nabla \cdot \hat{\mathbf{u}} + \nabla (\bar{u} H) \cdot \hat{\mathbf{u}} &= f \\ \bar{u} H \nabla \cdot \hat{\mathbf{u}} + (H \nabla \bar{u} + \bar{u} \nabla H) \cdot \hat{\mathbf{u}} &= f. \end{aligned}$$

After a little rearranging, this produces

$$\begin{aligned} H \hat{\mathbf{u}} \cdot \nabla \bar{u} + (H \nabla \cdot \hat{\mathbf{u}} + \nabla H \cdot \hat{\mathbf{u}}) \bar{u} &= f \\ H \hat{\mathbf{u}} \cdot \nabla \bar{u} + \nabla \cdot (H \hat{\mathbf{u}}) \bar{u} &= f. \end{aligned} \quad (13.16)$$

Next, because this system includes one equation and three unknowns, two are eliminated by *prescribing* the direction of flow. In the absence of surface velocity data, one choice for this direction is down the gradient of *driving stress* $\boldsymbol{\tau}_d = [\tau_x \ \tau_y \ 0]^\top$, derived by integrating vertically the forcing term of first-order momentum balance (9.25):

$$\boldsymbol{\tau}_d = \int_B^S \rho g \nabla S \ dz = \rho g H \nabla S. \quad (13.17)$$

Upon closer inspection of balance velocity relation (13.12), and using the fact that the partial derivative of the vertically integrated velocity with respect to z is zero,

$$\nabla \cdot (H\bar{\mathbf{u}}) = \nabla \cdot \left(\int_B^S \mathbf{u} dz \right) = \frac{\partial}{\partial x} \left(\int_B^S \mathbf{u} dz \right) + \frac{\partial}{\partial y} \left(\int_B^S \mathbf{u} dz \right).$$

Hence only the two horizontal components \hat{u} and \hat{v} of balance velocity (13.1) remain, and the domain is reduced to $\Omega \in \mathbb{R}^2$. Additionally, because only the *direction* of ice flow is imposed, the direction ∇S may be used in place of full driving stress (13.17).

Let the direction of imposed flow be defined as

$$\hat{\mathbf{u}} = \frac{\mathbf{d}}{\|\mathbf{d}\|}, \quad \mathbf{d} = \begin{bmatrix} d_x \\ d_y \end{bmatrix}. \quad (13.18)$$

Due to the fact that the solution to balance-velocity Equation (13.15, 13.13) will be highly sensitive to variations in the data used to define (13.18), an additional term is added to flow-direction (13.18) in such a way that direction is decomposed into data \mathbf{d}^{data} and Laplace-blurring \mathbf{d}^s terms (Brinkerhoff and Johnson, 2015)

$$\mathbf{d} = \mathbf{d}^{\text{data}} + \mathbf{d}^s, \quad \mathbf{d}^s = (\kappa H)^2 \nabla \cdot (\nabla \mathbf{d}),$$

where the constant κ adjusts the amount of smoothing proportional to the ice thickness H . The expansion of this linear system is

$$\begin{bmatrix} d_x \\ d_y \end{bmatrix} - (\kappa H)^2 \begin{bmatrix} \frac{\partial^2 d_x}{\partial x^2} + \frac{\partial^2 d_x}{\partial y^2} \\ \frac{\partial^2 d_y}{\partial x^2} + \frac{\partial^2 d_y}{\partial y^2} \end{bmatrix} = \begin{bmatrix} d_x^{\text{data}} \\ d_y^{\text{data}} \end{bmatrix},$$

thus illuminating two equations for the unknowns d_x and d_y ,

$$d_x - (\kappa H)^2 \left(\frac{\partial^2 d_x}{\partial x^2} + \frac{\partial^2 d_x}{\partial y^2} \right) = d_x^{\text{data}} \quad (13.19)$$

$$d_y - (\kappa H)^2 \left(\frac{\partial^2 d_y}{\partial x^2} + \frac{\partial^2 d_y}{\partial y^2} \right) = d_y^{\text{data}}. \quad (13.20)$$

13.1.1 Variational forms for \mathbf{d}

The variational problem associated with smoothed driving stress equations (13.19, 13.20) reads: find $d_x \in M^h \subset L^2(\Omega)$ (see L^2 space (1.13)) such that

$$\begin{aligned} \int_\Omega d_x \phi \ d\Omega - \int_\Omega (\kappa H)^2 \left(\frac{\partial^2 d_x}{\partial x^2} + \frac{\partial^2 d_x}{\partial y^2} \right) \phi \ d\Omega &= \int_\Omega d_x^{\text{data}} \phi \ d\Omega \\ + \int_\Omega d_x \phi \ d\Omega + \int_\Omega (\kappa H)^2 (\nabla d_x)_h \cdot (\nabla \phi)_h \ d\Omega \\ - \int_\Gamma \phi (\kappa H)^2 (\nabla d_x)_h \cdot \mathbf{n}_h \ d\Gamma &= \int_\Omega d_x^{\text{data}} \phi \ d\Omega \end{aligned}$$

for all $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ (see test space (1.11)); and $d_y \in M^h \subset L^2(\Omega)$ such that

$$\begin{aligned} \int_{\Omega} d_y \psi \, d\Omega - \int_{\Omega} (\kappa H)^2 \left(\frac{\partial^2 d_y}{\partial x^2} + \frac{\partial^2 d_y}{\partial y^2} \right) \psi \, d\Omega &= \int_{\Omega} d_y^{\text{data}} \psi \, d\Omega \\ \int_{\Omega} d_y \psi \, d\Omega + \int_{\Omega} (\kappa H)^2 (\nabla d_y)_h \cdot (\nabla \psi)_h \, d\Omega \\ - \int_{\Gamma} \psi (\kappa H)^2 (\nabla d_y)_h \cdot \mathbf{n}_h \, d\Gamma &= \int_{\Omega} d_y^{\text{data}} \psi \, d\Omega \end{aligned}$$

for all $\psi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$. Horizontal vector components are denoted $\mathbf{v}_h = [v_x \, v_y]^T$.

13.2 The magnitude of flowing ice

On closer examination of velocity-balance relation (13.16), the equation is seen to be of the same form as the advection-reaction equation described in §5.5,

$$\mathbf{a} \cdot \nabla u + su = f,$$

with unknown quantity $u = \bar{u}$, velocity $\mathbf{a} = H\hat{\mathbf{u}}$, and reaction coefficient $s = \nabla \cdot (H\hat{\mathbf{u}})$. As previously discussed in §5.5, equations of this type suffer from numerical oscillations requiring the use of stabilization. Therefore, defining the linear differential operator associated with problem (13.16),

$$\mathcal{L}u = H\hat{\mathbf{u}} \cdot \nabla \bar{u} + \nabla \cdot (H\hat{\mathbf{u}})\bar{u}, \quad (13.21)$$

the stabilized form is stated using general stabilized form (5.16) with test function $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ and intrinsic-time parameter τ_{BV} ,

$$(\phi, \mathcal{L}\bar{u}) + (\mathbb{L}\phi, \tau_{\text{BV}}(\mathcal{L}\bar{u} - f)) = (\phi, f), \quad (13.22)$$

where operator \mathbb{L} is a differential operator typically chosen from

$$\mathbb{L} = +\mathcal{L} \quad \text{Galerkin/least-squares (GLS)} \quad (13.23)$$

$$\mathbb{L} = +\mathcal{L}_{\text{adv}} \quad \text{SUPG} \quad (13.24)$$

$$\mathbb{L} = -\mathcal{L}^* \quad \text{subgrid-scale model (SSM)} \quad (13.25)$$

with $\mathcal{L}_{\text{adv}} = H\hat{\mathbf{u}} \cdot \nabla \bar{u}$, the advective part of the operator \mathcal{L} .

An appropriate stabilization parameter for this problem is given by (5.22), as derived by Codina, 1998. After making the appropriate substitutions to ADR parameter (5.22), the intrinsic-time parameter is

$$\tau_{\text{BV}} = \frac{1}{\frac{2H}{h} + \nabla \cdot (H\hat{\mathbf{u}})}, \quad (13.26)$$

with cell diameter h . Here, the fact that $\|\hat{\mathbf{u}}\| = 1$ and that there is no diffusion present has been used.

13.2.1 Variational form for \bar{u}

Using intrinsic-time parameter (13.26) and operator (13.21), the variational problem associated with (13.15, 13.13) reads: find $\bar{u} \in M^h \subset L^2(\Omega)$ such that

$$\mathcal{B}(\phi, \bar{u}) = \ell(\phi), \quad (13.27)$$

where

$$\begin{aligned} \mathcal{B}(\phi, \bar{u}) &= (\phi, \mathcal{L}\bar{u}) + (\mathbb{L}\phi, \tau_{\text{BV}}(\mathcal{L}\bar{u})) \\ \ell(\phi) &= (\phi, f) + (\mathbb{L}\phi, \tau_{\text{BV}}f), \end{aligned}$$

for all $\phi \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ and \mathbb{L} given by one of (13.23), (13.24), or (13.25).

The implementation of this problem by CSLVR is shown in Code Listing 13.1.

Code Listing 13.1: CSLVR implementation of the BalanceVelocity class.

```
from fenics import *
from dolfin_adjoint import *
from cs_lvr.physics import Physics
from cs_lvr.d2model import D2Model
from cs_lvr.io import print_text, print_min_max
import numpy as np
import sys

class BalanceVelocity(Physics):
    """
    Balance velocity solver.

    Class representing balance velocity physics.

    Use like this:

    >>> bv = BalanceVelocity(model, 5.0)
    ::: INITIALIZING VELOCITY-BALANCE PHYSICS :::
    >>> bv.solve()
    ::: solving BalanceVelocity :::
    ::: calculating surface gradient :::
    Process 0: Solving linear system of size 9034 x 9034 (PETSc Krylov solver).
    Process 0: Solving linear system of size 9034 x 9034 (PETSc Krylov solver).
    dSdx <min, max> : <-1.107e+00, 8.311e-01>
    dSdy <min, max> : <-7.928e-01, 1.424e+00>
    ::: solving for smoothed x-component of driving stress with kappa = 5.0 :::
    Process 0: Solving linear variational problem.
    Nx <min, max> : <-1.607e+05, 3.628e+05>
    ::: solving for smoothed y-component of driving stress :::
    Process 0: Solving linear variational problem.
    Ny <min, max> : <-2.394e+05, 2.504e+05>
    ::: calculating normalized velocity direction from driving stress :::
    dx <min, max> : <-1.000e+00, 9.199e-01>
    dy <min, max> : <-9.986e-01, 1.000e+00>
    ::: solving velocity balance magnitude :::
    Process 0: Solving linear variational problem.
    Ubar <min, max> : <-5.893e+03, 9.844e+03>
    ::: removing negative values of balance velocity :::
    Ubar <min, max> : <0.000e+00, 9.844e+03>

    Args:
    :model: a :class:`~d2model.D2Model` instance holding all pertinent
            variables.
    :kappa: a floating-point value representing surface smoothing
            radius in units of ice thickness :math:'H = S-B'.

    Returns:
    text printed to the screen.
    """
    def __init__(self, model, kappa=5.0, stabilization_method='SUPG'):
        """
        balance velocity init.
        """
        self.kappa = kappa

        s = """::: INITIALIZING VELOCITY-BALANCE PHYSICS :::
        print_text(s, cls=self)

        if type(model) != D2Model:
            s = """>>> BalanceVelocity REQUIRES A 'D2Model' INSTANCE, NOT %s <<<
            print_text(s % type(model), 'red', 1)
            sys.exit(1)

        Q = model.Q
        S = model.S
        B = model.B
        H = S - B
        h = model.h
        N = model.N
        uhat = model.uhat
        vhat = model.vhat
        adot = model.adot
        Fb = model.Fb

        # =====
        # form to calculate direction of flow (down driving stress gradient) :
        phi = TestFunction(Q)
        ubar = TrialFunction(Q)
        kappa = Constant(kappa)

        # stabilization test space :
        Uhat = as_vector([uhat, vhat])
        tau = 1 / (2 * H / h + div(H * Uhat))
        phihat = phi + tau * dot(Uhat, grad(phi))

        # the left-hand side :
        def L(u): return u * H * div(Uhat) + dot(grad(u * H), Uhat)
        def L_star(u): return u * H * div(Uhat) - dot(grad(u * H), Uhat)
        def L_adv(u): return dot(grad(u * H), Uhat)

        Nb = sqrt(B.dxdx**2 + B.dxdy**2 + 1)
        Ns = sqrt(S.dxdx**2 + S.dxdy**2 + 1)
        f = Ns * adot - Nb * Fb

        # use streamline-upwind/Petrov-Galerkin :
        if stabilization_method == 'SUPG':
            s = """using streamline-upwind/Petrov-Galerkin stabilization -"""
            self.B = L(ubar) * phi * dx \
                + inner(L_adv(phi), tau * L(ubar)) * dx
            self.a = f * phi * dx \
                + inner(L_adv(phi), tau * f) * dx
```

13.3 Continent-wide simulations

Solutions to balance-velocity variational problem (13.27) were obtained over the entire continents of Antarctica and Greenland for each of stabilization schemes (13.23), (13.24), and (13.25). In order to complete these simulations, some assumptions had to be made regarding forcing term (13.13).

First, we assumed that $w(S) - w(B) = 0$ everywhere. While this is assumption is not true in general, it has been our experience from investigation of velocity solutions obtained via the methods of Chapter 12 that the vertical velocity remains relatively constant throughout the vertical coordinate over much of the ice-sheet domain, except areas where the velocity changes direction abruptly. Second, we assumed that the ice sheet thickness is in equilibrium, and hence $\partial_t H = 0$. Finally, we did not prescribe any basal melting, and thus $F_b = 0$.

To create the finite-element mesh, we incorporated the dynamic version of GMSH (Geuzaine and Remacle, 2009) into CSLVR. This software allows us to generate finite-element meshes with cell diameters set by any function we wish. Following the work of Brinkerhoff and Johnson (2015), we generated meshes for each ice-sheet with cell diameter h given by

$$h = mH$$

for some constant m .

Finally, simulations were run using flow directions \mathbf{d}^{data} both in the down-surface-gradient direction (13.18) and in the direction of surface observations \mathbf{u}_{ob} (Figures 13.1 and 13.6). Due to the fact that the surface gradient is nearly zero over the floating shelves of Antarctica, the flow direction resulting from the surface gradient is not well defined. To investigate this problem, we applied the direction of flow to be in the direction of surface velocity observations $\mathbf{u}_{ob} = [u_{ob} \ v_{ob}]^T$ in these areas, with improved results (Figure 13.8). For comparison purposes, we then ran a simulation over Antarctica imposing the direction of flow entirely in the direction of observations \mathbf{u}_{ob} , and noticed that the results obtained without smoothing contained high error in regions without velocity observations (Figure 13.9). To remedy this, we ran one final test over Antarctica with direction of flow imposed in the direction of velocity observations \mathbf{u}_{ob} where observations are present and down the surface gradient ∇S where they are not (Figure 13.10).

Results indicate that the vertically averaged flow field $\bar{\mathbf{u}}$ roughly reproduces the general pattern of recorded surface velocities over both Greenland (Figures 13.2 and 13.3) and Antarctica (Figures 13.7 and 13.9). The difference between the vertically-averaged velocity and the recorded surface speed provides some insight into the variation of velocity with depth (Figures 13.4, 13.5, 13.11, 13.12, 13.13, and 13.14). That is, in regions where the misfit is high, we expect significant differences between the surface and the basal velocity. However, due to the fact that the data – including topography, accumulation, $\partial_t H$ and F_b – contain a high degree of uncertainty, we expect that with better data the sporadic variance structure evident here will be diminished.

```
# use Galerkin/least-squares
elif stabilization_method == 'GLS':
    s = " - using Galerkin/least-squares stabilization -"
    self.B = L(u_bar) * phi * dx \
        + inner(L(phi), tau*L(u_bar)) * dx
    self.a = f * phi * dx \
        + inner(L(phi), tau*f) * dx

# use subgrid-scale-model :
elif stabilization_method == 'SSM':
    s = " - using subgrid-scale-model stabilization -"
    self.B = L(u_bar) * phi * dx \
        + inner(L_star(phi), tau*L(u_bar)) * dx
    self.a = f * phi * dx \
        + inner(L_star(phi), tau*f) * dx

print_text(s, cls=self)

def solve_direction_of_flow(self, d, annotate=False):
    r"""
    Solve for the direction of flow, attained in two steps :

    1. Solve for the smoothed components of :d: :

    .. math::

        \mathbf{d}_s = \frac{1}{H} \nabla S \cdot \nabla \mathbf{d} \quad \nabla \mathbf{d} = \frac{1}{H} \nabla S \cdot \nabla \mathbf{d}

    for components :math:'d_x' and :math:'d_y' saved respectively
    to 'model.d_x' and 'model.d_y'.

    2. Calculate the normalized flux directions :

    .. math::

        \hat{u} = \frac{d_x}{\sqrt{d_x^2 + d_y^2}} \quad \hat{v} = \frac{d_y}{\sqrt{d_x^2 + d_y^2}}

    saved respectively to 'model.uhat' and 'model.vhat'.
    """
    model = self.model
    Q = model.Q
    S = model.S
    B = model.B
    H = S - B
    N = model.N
    phi = TestFunction(Q)
    d_x = TrialFunction(Q)
    d_y = TrialFunction(Q)
    kappa = Constant(self.kappa)

    # horizontally smoothed direction of flow :
    a_dSdx = d_x * phi * dx \
        + (kappa*H)**2 * dot(grad(phi), grad(d_x)) * dx \
        - (kappa*H)**2 * dot(grad(d_x), N) * phi * ds
    L_dSdx = d[0] * phi * dx

    a_dSdy = d_y * phi * dx \
        + (kappa*H)**2 * dot(grad(phi), grad(d_y)) * dx \
        - (kappa*H)**2 * dot(grad(d_y), N) * phi * ds
    L_dSdy = d[1] * phi * dx

    # update velocity direction :
    s = " :: solving for smoothed x-component of flow direction " + \
        "with kappa = %g ::" % self.kappa
    print_text(s, cls=self)
    solve(a_dSdx == L_dSdx, model.d_x, annotate=annotate)
    print_min_max(model.d_x, 'd_x', cls=self)

    s = " :: solving for smoothed y-component of flow direction " + \
        "with kappa = %g ::" % self.kappa
    print_text(s, cls=self)
    solve(a_dSdy == L_dSdy, model.d_y, annotate=annotate)
    print_min_max(model.d_y, 'd_y', cls=self)

    # normalize the direction vector :
    s = " :: calculating normalized flux direction from \nabla S ::"
    print_text(s, cls=self)
    d_x_v = model.d_x.vector().array()
    d_y_v = model.d_y.vector().array()
    d_n_v = np.sqrt(d_x_v**2 + d_y_v**2 + 1e-16)
    model.assign_variable(model.uhat, d_x_v / d_n_v, cls=self)
    model.assign_variable(model.vhat, d_y_v / d_n_v, cls=self)

def solve(self, annotate=False):
    r"""
    Solve the balance velocity magnitude :math:'\nabla \mathbf{u}' \cdot \nabla \mathbf{u}'.

    This will be completed in three steps,

    1. Solve for the smoothed component of surface gradient :

    .. math::

        d_x = \frac{\partial S}{\partial x} \quad d_y = \frac{\partial S}{\partial y}

    saved respectively to 'model.d_x' and 'model.d_y'.

    2. Calculate the normalized flux directions :

    .. math::

        \hat{u} = \frac{d_x}{\sqrt{d_x^2 + d_y^2}} \quad \hat{v} = \frac{d_y}{\sqrt{d_x^2 + d_y^2}}

    saved respectively to 'model.d_x' and 'model.d_y'.

    3. Calculate the balance velocity magnitude
    :math:'\nabla \mathbf{u}' \cdot \nabla \mathbf{u}'.

    .. math::

        \nabla S \cdot \nabla \mathbf{u} = f

    saved to 'model.Ubar'.
    """
    model = self.model

    # calculate balance-velocity :
    s = " :: solving velocity balance magnitude ::"
    print_text(s, cls=self)
    solve(self.B == self.a, model.Ubar, annotate=annotate)
    print_min_max(model.Ubar, 'Ubar', cls=self)

    # enforce positivity of balance-velocity :
    s = " :: removing negative values of balance velocity ::"
    print_text(s, cls=self)
    Ubar_v = model.Ubar.vector().array()
    Ubar_v[Ubar_v < 0] = 0
    model.assign_variable(model.Ubar, Ubar_v, cls=self)
```

Defining the direction of flow in the direction of velocity observations over the shelves of Antarctica is an improvement over the down-surface-gradient direction (compare Figures 13.7 and 13.8). However, there appear to remain substantial differences between these two velocities in these regions (Figure 13.12). This may be due to substantial basal melt or freeze-on under the shelves, implying in this case that our specification of $F_b = 0$ is entirely inappropriate. Additionally, the balance velocity changes from under-estimation to over-estimation of the surface velocity over the fastest-moving parts of interior regions when a meaningful direction of flow is imposed over floating-ice regions (compare Figures 13.11 and 13.12). This illuminates the fact that the balance velocity of the floating-ice shelves has at least some affect on the velocity deep within the interior.

Filling in the gaps of missing velocity data \mathbf{u}_{ob} with $-\nabla S$ appeared to have the most effect on results derived without smoothing (Figure 13.10 and 13.14).

Finally, the solutions attained with the stabilization schemes 13.23), (13.24), and (13.25) varied most with lower values of smoothing radius κ . At the lowest κ -values, the GLS model appears to on-average under estimate the velocity, while the SSM model seems to on-average over estimate. Remarkably, all calculations over estimate the velocity of South-West Greenland (Figures 13.4 and 13.5) and match the deep-interior velocity observations of Antarctica relatively well (Figures 13.11, 13.12, 13.13, and 13.14).

13.3.1 Greenland

The Greenland simulations used topography S and B given by Bamber et al. (2013), and an accumulation/ablation function \dot{a} provided by Veen et al. (2001); Burgess et al. (2010).

The CSLVR scripts used to generate the mesh, data, and perform the simulation are shown in Code Listings 13.2, 13.3, and 13.4, respectively.

Table 13.1: Greenland \bar{u} simulation variables.

Variable	Value	Units	Description
$\partial_t H$	0	m a^{-1}	H obs. rate of change
F_b	0	m a^{-1}	basal water discharge
m	5	—	mesh refinement
N_e	92855	—	number of cells
N_n	49064	—	number of vertices

Code Listing 13.2: CSLVR script used to generate the 2D mesh for Greenland.

```
from cslvr import *

kappa = 5.0 # ice thickness to refine

# data preparation :
out_dir = 'dump/meshes/'
mesh_name = 'greenland_2D_%iH_mesh' % int(kappa)

# get the data :
bamber = DataFactory.get_bamber()
rignot = DataFactory.get_rignot()

# process the data :
dbm = DataInput(bamber, gen_space=False)
drg = DataInput(rignot, gen_space=False)
drg.change_projection(dbm)
```

```
# form field from which to refine :
dbm.data['ref'] = kappa*dbm.data['H'].copy()
dbm.data['ref'][dbm.data['ref'] < kappa*1000.0] = kappa*1000.0

# nice to plot the refinement field to check that you're doing what you want :
# plotice(dbm, 'ref', name='ref', direc=out_dir,
#         title='ref', cmap='viridis',
#         show=False, scale='lin', tp=False, cb_format='%1e')

# generate the contour :
m = MeshGenerator(dbm, mesh_name, out_dir)

m.create_contour('mask', zero_ctr=0.99, skip_pts=10)
# m.create_contour('H', zero_ctr=200, skip_pts=5)

m.eliminate_intersections(dist=200)
# m.transform_contour(rignot)
# m.check_dist()
m.write_gmsh_contour(boundary_extend=False)
# m.plot_contour()
m.close_file()

# refine :
ref_bm = MeshRefiner(dbm, 'ref', gmsh_file_name = out_dir + mesh_name)

a.aid = ref_bm.add_static_attractor()
ref_bm.set_background_field(aid)

# finish stuff up :
ref_bm.finish(gui = False, out_file_name = out_dir + mesh_name)
ref_bm.convert_msh_to_xml()
```

Code Listing 13.3: CSLVR script used to generate the data used by the Greenland balance velocity calculation of Code Listing 13.4.

```
from cslvr import *

thklim = 1e-2
mesh_H = 5

# collect the raw data :
searise = DataFactory.get_searise(thklim)
bamber = DataFactory.get_bamber(thklim)
rignot = DataFactory.get_rignot()

# set the output directory :
out_dir = 'dump/vars/'

# load a mesh :
mesh = Mesh('dump/meshes/greenland_2D_%iH_mesh.xml.gz' % mesh_H)

# create data objects to use with varglas :
dsr = DataInput(searise, mesh=mesh)
dbm = DataInput(bamber, mesh=mesh)
drg = DataInput(rignot, mesh=mesh)

# Rignot dataset is on a different projection :
drg.change_projection(dbm)

B = dbm.get_expression("B", near=False)
S = dbm.get_expression("S", near=False)
M = dbm.get_expression("mask", near=True)
adot = dsr.get_expression("adot", near=False)
u_ob = drg.get_expression("vx", near=False)
v_ob = drg.get_expression("vy", near=False)
U_msk = drg.get_expression("mask", near=True)

model = D2Model(mesh, out_dir = 'results/')
model.calculate_boundaries(mask=M, U_mask=U_msk, adot=adot)

# calculate_boundaries() initializes model.mask, model.U_mask, and model.adot

model.init_S(S)
model.init_B(B)
model.init_U_ob(u_ob, v_ob)

lst = [model.S,
        model.B,
        model.adot,
        model.mask,
        model.u_ob,
        model.v_ob,
        model.U_mask]

f = HDF5File(mpi_comm_world(), out_dir + 'state.h5', 'w')

model.save_list_to_hdf5(lst, f)
model.save_subdomain_data(f)
model.save_mesh(f)

f.close()

model.save_xdmf(model.cf, 'cf')
```

Code Listing 13.4: CSLVR script used to calculate the balance velocity for Greenland.

```
from cslvr import *

mesh_H = 5

# set plot directory :
plt_dir = '.../images/balance_velocity/greenland/d_U_ob/'

# load the data :
f = HDF5File(mpi_comm_world(), 'dump/vars/state.h5', 'r')

# the balance velocity uses a 2D-model :
model = D2Model(f, out_dir = 'results/')

# set the calculated subdomains :
model.set_subdomains(f)
```

```
# use the projection of the dataset 'searise' for plotting :
searise = DataFactory.get_searise()

model.init_S(f)
model.init_B(f)
model.init_adot(f)
model.init_mask(f)
model.init_U_ob(f,f)
model.init_U_mask(f)

kappas = [0.5,10]
methods = ['SUPG', 'SSM', 'GLS']

# the imposed direction of flow :
d = (model.u_ob, model.v_ob)
#d = (-model.S.dx(0), -model.S.dx(1))

## plot the observed surface speed :
#U_max = model.U_ob.vector().max()
#U_min = model.U_ob.vector().min()
#U_lvls = array([U_min, 2, 10, 20, 50, 100, 200, 500, 1000, U_max])
#plotIce(searise, model.U_ob, name='U_ob', direc=plt_dir,
#        title='U Ob', cmap='viridis',
#        show=False, levels=U_lvls, tp=False, cb_format='%1.1e')

for kappa in kappas:
    for method in methods:

        bv = BalanceVelocity(model, kappa=kappa, stabilization_method=method)
        bv.solve_direction_of_flow(d)
        bv.solve()

        U_max = model.Ubar.vector().max()
        U_min = model.Ubar.vector().min()
        U_lvls = array([U_min, 2, 10, 20, 50, 100, 200, 500, 1000, U_max])

        name = 'Ubar_{iH}_kappa_{i}%s' % (mesh_H, kappa, method)
        tit = r'$\bar{u}_{\{i\}}$' % kappa
        plotIce(searise, model.Ubar, name=name, direc=plt_dir,
                title=tit, cmap='viridis',
                show=False, levels=U_lvls, tp=False, cb_format='%1.1e')

# calculate the misfit
misfit = Function(model.Q)
Ubar_v = model.Ubar.vector().array()
U_ob_v = model.U_ob.vector().array()
m_v = U_ob_v - Ubar_v
model.assign_variable(misfit, m_v)

m_max = misfit.vector().max()
m_min = misfit.vector().min()
m_lvls = array([m_min, -50, -10, -5, -1, 1, 5, 10, 50, m_max])

name = 'misfit_{iH}_kappa_{i}%s' % (mesh_H, kappa, method)
tit = r'$M_{\{i\}}$' % kappa
plotIce(searise, misfit, name=name, direc=plt_dir,
        title=tit, cmap='RdGy',
        show=False, levels=m_lvls, tp=False, cb_format='%1.1e')
```

13.3.2 Antarctica

The Antarctica simulations used topography S and B provided by Fretwell et al. (2013), and an accumulation/ablation function \dot{a} provided by Arthern, Winebrenner, and Vaughan (2006); Le Brocq, Payne, and Vieli (2010).

The CSLVR scripts used to generate the mesh, data, and perform the simulation are shown in Code Listings 13.5, 13.6, and 13.7, respectively.

Table 13.2: Antarctica \bar{u} simulations variables.

Variable	Value	Units	Description
$\partial_t H$	0	m a^{-1}	H obs. rate of change
F_b	0	m a^{-1}	basal water discharge
m	10	—	mesh refinement
N_e	162211	—	number of cells
N_n	82894	—	number of vertices

Code Listing 13.5: CSLVR script used to generate the 2D mesh for Antarctica.

```
from cslvr import *

kappa = 10.0 # ice thickness to refine

# data preparation :
out_dir = 'dump/meshes/'
mesh_name = 'antarctica_2D_{iH}_mesh' % int(kappa)

# get the data :
bedmap2 = DataFactory.get_bedmap2()

# process the data :
db2 = DataInput(bedmap2, gen_space=False)

db2.set_data_val("H", 32767, 0.0)
db2.set_data_val("S", 32767, 0.0)
```

```
# form field from which to refine :
db2.data['ref'] = kappa*db2.data['H'].copy()
db2.data['ref'][db2.data['ref'] < kappa*1000.0] = kappa*1000.0

## nice to plot the refinement field to check that you're doing what you want :
#plotIce(db2, 'ref', name='ref', direc=out_dir,
#        title='ref', cmap='viridis',
#        show=False, scale='lin', tp=False, cb_format='%1.1e')

# generate the contour :
m = MeshGenerator(db2, mesh_name, out_dir)

m.create_contour('mask', zero_ctr=0.99, skip_pts=10)
#m.create_contour('H', zero_ctr=200, skip_pts=5)

m.eliminate_intersections(dist=200)
m.write_gmsh_contour(boundary_extend=False)
#m.plot_contour()
m.close_file()

# refine :
ref_b2 = MeshRefiner(db2, 'ref', gmsh_file_name = out_dir + mesh_name)

a,aid = ref_b2.add_static_attractor()
ref_b2.set_background_field(aid)

# finish stuff up :
ref_b2.finish(gui = False, out_file_name = out_dir + mesh_name)
ref_b2.convert_msh_to_xml()
```

Code Listing 13.6: CSLVR script used to generate the data used by the Antarctica balance velocity calculation of Code Listing 13.7.

```
from cslvr import *

thklim = 1.0
mesh_H = 10

# collect the raw data :
bedmap2 = DataFactory.get_bedmap2(thklim=thklim)
bedmap1 = DataFactory.get_bedmap1(thklim=thklim)
measures = DataFactory.get_ant_measures(res=900)

# set the output directory :
out_dir = 'dump/vars/'

# load a mesh :
mesh = Mesh('dump/meshes/antarctica_2D_{iH}_mesh.xml.gz' % mesh_H)

# create data objects to use with varglas :
db1 = DataInput(bedmap1, mesh=mesh)
db2 = DataInput(bedmap2, mesh=mesh)
dbm = DataInput(measures, mesh=mesh)

S = db2.get_expression("S", near=False)
B = db2.get_expression("B", near=False)
M = db2.get_expression("mask", near=True)
adot = db1.get_expression("acca", near=False)
u_ob = dbm.get_expression("vy", near=False)
v_ob = dbm.get_expression("vy", near=False)
U_msk = dbm.get_expression("mask", near=True)

model = D2Model(mesh, out_dir = 'results/')
model.calculate_boundaries(mask=M, U_mask=U_msk, adot=adot)

# calculate_boundaries() initializes model.mask, model.U_mask, and model.adot

model.init_S(S)
model.init_B(B)
model.init_U_ob(u_ob, v_ob)

lst = [model.S,
        model.B,
        model.adot,
        model.mask,
        model.u_ob,
        model.v_ob,
        model.U_mask]

f = HDF5File(mpi_comm_world(), out_dir + 'state.h5', 'w')

model.save_list_to_hdf5(lst, f)
model.save_subdomain_data(f)
model.save_mesh(f)

f.close()
```

Code Listing 13.7: CSLVR script used to calculate the balance velocity for Antarctica.

```
from cslvr import *

mesh_H = 10

# set plot directory :
plt_dir = '../..../images/balance_velocity/antarctica/'

# load the data :
f = HDF5File(mpi_comm_world(), 'dump/vars/state.h5', 'r')

# the balance velocity uses a 2D-model :
model = D2Model(f, out_dir = 'results/')

# set the calculated subdomains :
model.set_subdomains(f)

# use the projection of the dataset 'bedmap1' for plotting :
bm1 = DataFactory.get_bedmap1()

model.init_S(f)
model.init_B(f)
model.init_adot(f)
model.init_mask(f)
model.init_U_ob(f,f)
model.init_U_mask(f)
```



```

# containers for the direction of flow :
d_U_ob_S_x = Function(model.Q)
d_U_ob_S_y = Function(model.Q)
d_gS_m_U_x = Function(model.Q)
d_gS_m_U_y = Function(model.Q)

# calculate the down-surface gradient :
dSdx = project(-model.S.dx(0))
dSdy = project(-model.S.dx(1))

# convert to numpy arrays :
dSdx_v = dSdx.vector().array()
dSdy_v = dSdy.vector().array()
d_U_ob_S_x_v = d_U_ob_S_x.vector().array()
d_U_ob_S_y_v = d_U_ob_S_y.vector().array()
d_gS_m_U_x_v = d_gS_m_U_x.vector().array()
d_gS_m_U_y_v = d_gS_m_U_y.vector().array()
u_ob_v = model.u_ob.vector().array()
v_ob_v = model.v_ob.vector().array()

# grounded: down grad(S) -- floating: U_ob :
d_U_ob_S_x_v[model.shf_dofs] = u_ob_v[model.shf_dofs]
d_U_ob_S_y_v[model.shf_dofs] = v_ob_v[model.shf_dofs]
d_U_ob_S_x_v[model.gnd_dofs] = dSdx_v[model.gnd_dofs]
d_U_ob_S_y_v[model.gnd_dofs] = dSdy_v[model.gnd_dofs]

# everywhere with U observations: U_ob -- everywhere without: down grad(S) :
d_gS_m_U_x_v[model.Uob_dofs] = u_ob_v[model.Uob_dofs]
d_gS_m_U_y_v[model.Uob_dofs] = v_ob_v[model.Uob_dofs]
d_gS_m_U_x_v[model.Uob_missing_dofs] = dSdx_v[model.Uob_missing_dofs]
d_gS_m_U_y_v[model.Uob_missing_dofs] = dSdy_v[model.Uob_missing_dofs]

# assign the numpy arrays back to the containers :
model.assign_variable(d_U_ob_S_x, d_U_ob_S_x_v)
model.assign_variable(d_U_ob_S_y, d_U_ob_S_y_v)
model.assign_variable(d_gS_m_U_x, d_gS_m_U_x_v)
model.assign_variable(d_gS_m_U_y, d_gS_m_U_y_v)

# the imposed direction of flow :
#d = (d_U_ob_S_x, d_U_ob_S_y)
#d = (d_gS_m_U_x, d_gS_m_U_y)
#d = (model.u_ob, model.v_ob)
d = (-model.S.dx(0), -model.S.dx(1))

## plot the observed surface speed :
#U_max = model.U_ob.vector().max()
#U_min = model.U_ob.vector().min()
#U_lvls = array([U_min, 2, 10, 20, 50, 100, 200, 500, 1000, U_max])
#plotIce(bm1, model.U_ob, name='U_ob', direc=plt_dir,
#        title=r'$\text{Vert } \mathbf{u}_{ob} \setminus \text{Vert}$', cmap='viridis',
#        show=False, levels=U_lvls, tp=False, cb_format='%.1e')

kappas = [0.5, 10]
methods = ['SUPG', 'SSM', 'GLS']

for kappa in kappas:
    for method in methods:
        bv = BalanceVelocity(model, kappa=kappa, stabilization_method=method)
        bv.solve_direction_of_flow(d)
        bv.solve()

        U_max = model.Ubar.vector().max()
        U_min = model.Ubar.vector().min()
        U_lvls = array([U_min, 2, 10, 20, 50, 100, 200, 500, 1000, U_max])

        name = 'Ubar_%iH_kappa_%i.%s' % (mesh_H, kappa, method)
        tit = r'$\bar{u}_{%i}$' % kappa
        plotIce(bm1, model.Ubar, name=name, direc=plt_dir,
                title=tit, cmap='viridis',
                show=False, levels=U_lvls, tp=False, cb_format='%.1e')

        # calculate the misfit
        misfit = Function(model.Q)
        Ubar_v = model.Ubar.vector().array()
        U_ob_v = model.U_ob.vector().array()
        m_v = U_ob_v - Ubar_v
        model.assign_variable(misfit, m_v)

        m_max = misfit.vector().max()
        m_min = misfit.vector().min()
        #m_lvls = array([m_min, -5e2, -1e2, -1e1, -1, 1, 1e1, 1e2, 5e2, m_max])
        m_lvls = array([m_min, -50, -10, -5, -1, 1, 5, 10, 50, m_max])

        name = 'misfit_%iH_kappa_%i.%s' % (mesh_H, kappa, method)
        tit = r'$M_{%i}$' % kappa
        plotIce(bm1, misfit, name=name, direc=plt_dir,
                title=tit, cmap='RdGy',
                show=False, levels=m_lvls, tp=False, cb_format='%.1e')

```

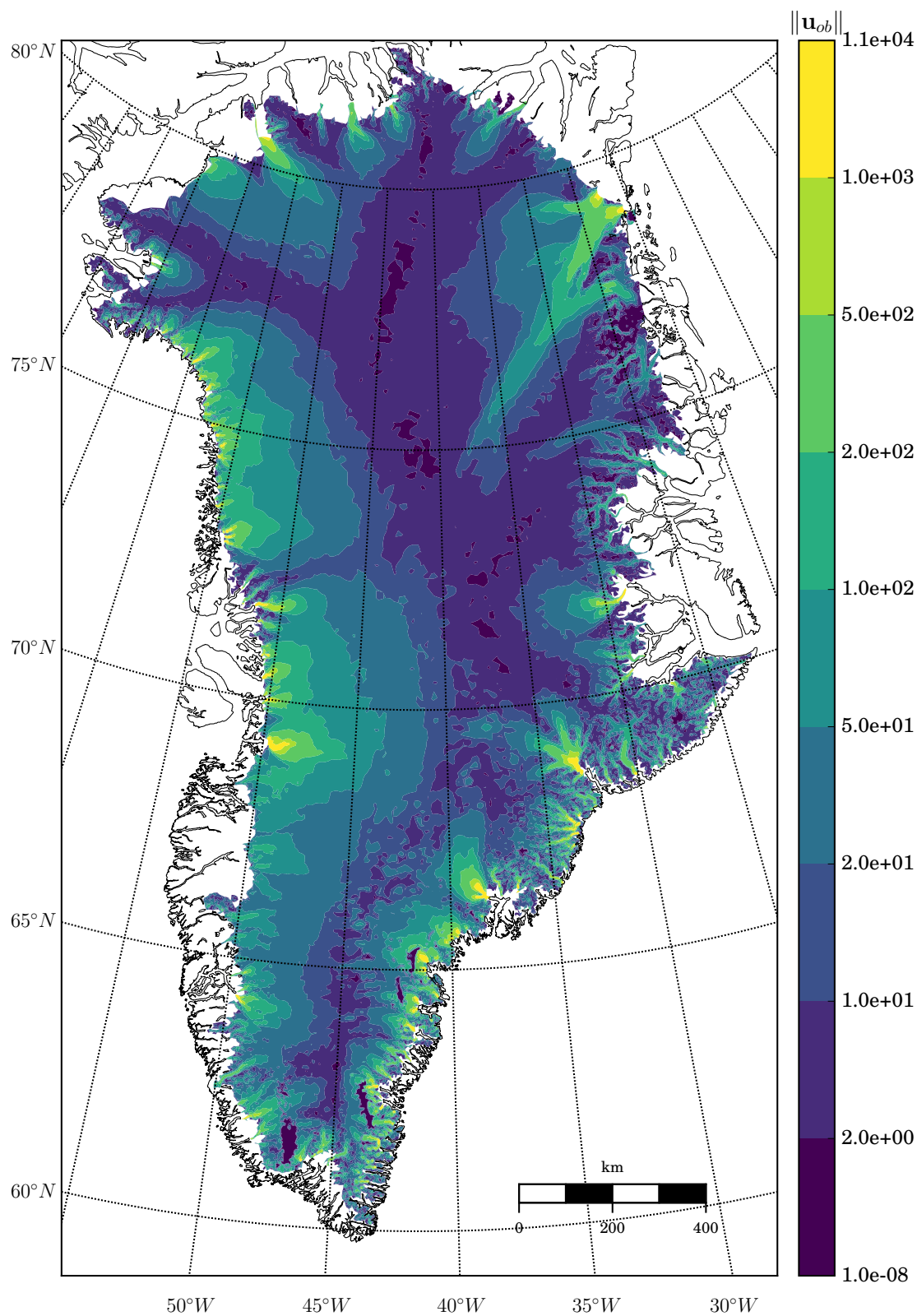


Figure 13.1: Surface velocity magnitude of Greenland for the polar year 2008–2009 provided by Rignot and Mouginot (2012).

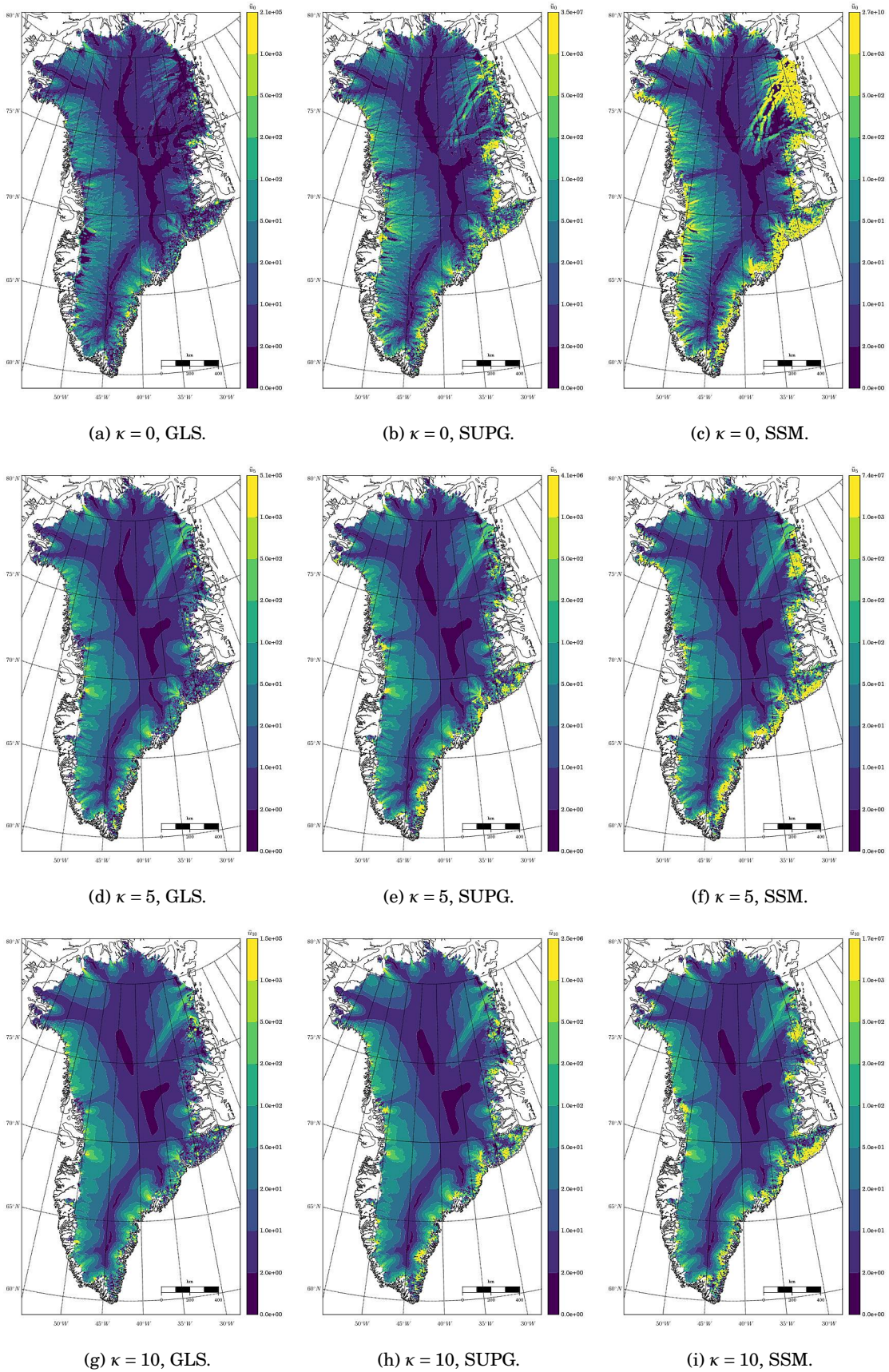


Figure 13.2: Balance velocity \bar{u} derived over Greenland with direction of flow imposed down the surface gradient ∇S , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23), streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24), or subgrid-scale-model (SSM) stabilization (13.25) in variational form (13.27).

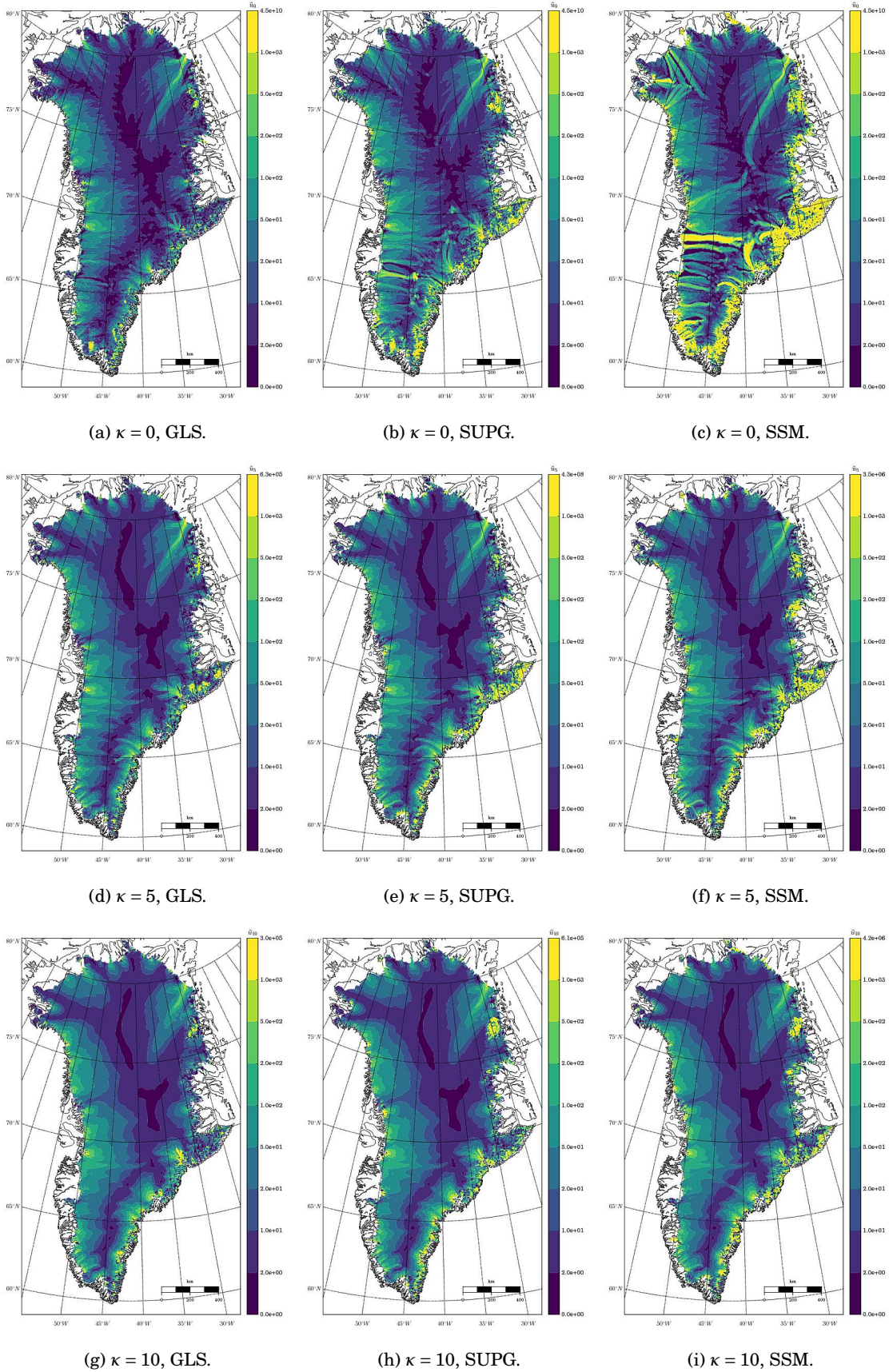


Figure 13.3: Balance velocity \bar{u} derived over Greenland with direction of flow imposed in the direction of surface velocity observations \mathbf{u}_{ob} , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23), streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24), or subgrid-scale-model (SSM) stabilization (13.25) in variational form (13.27).

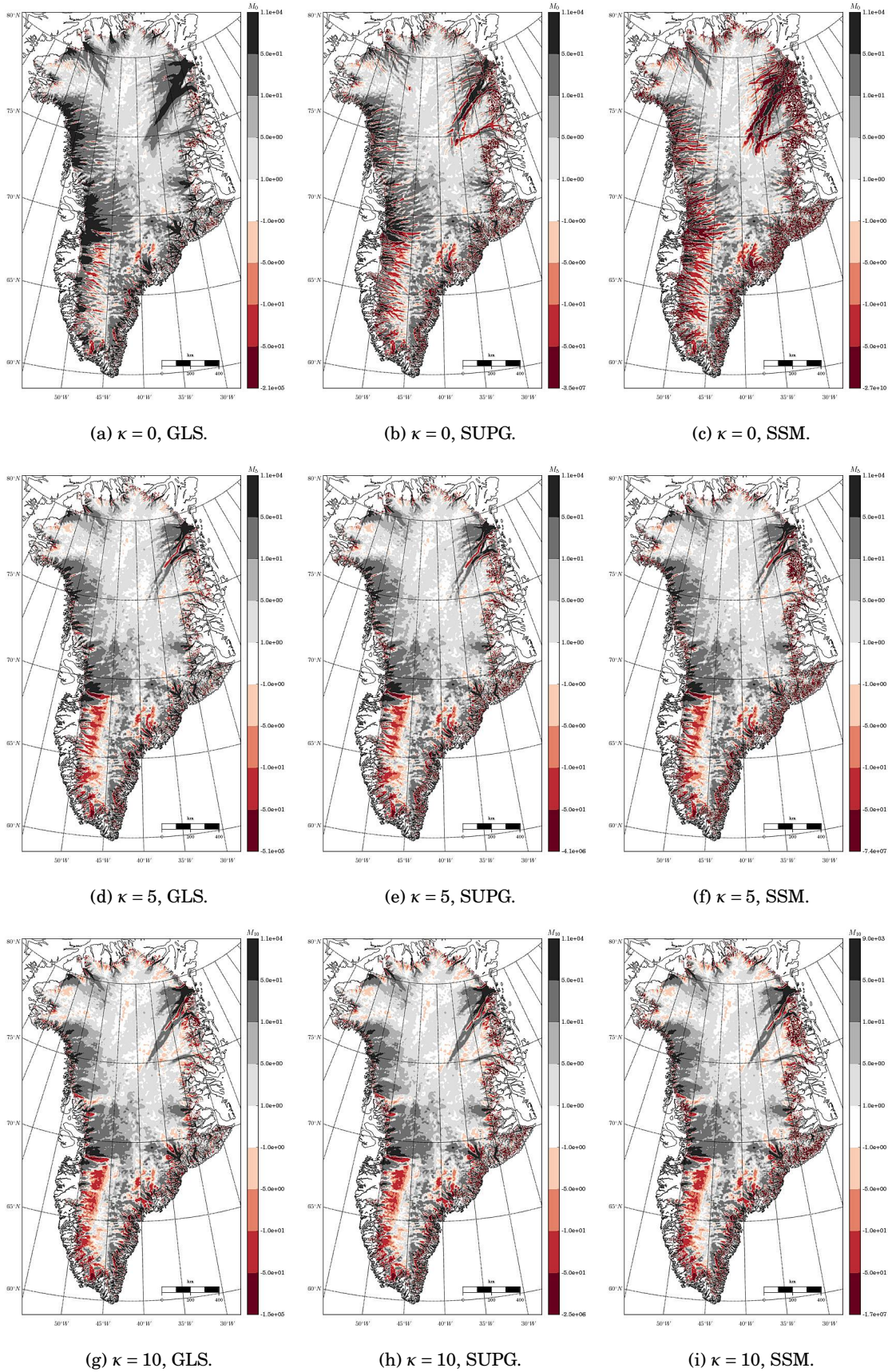


Figure 13.4: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} derived over Greenland with imposed direction of flow down the surface gradient VS, where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23), streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24), or subgrid-scale-model (SSM) stabilization (13.25) in variational form (13.27).

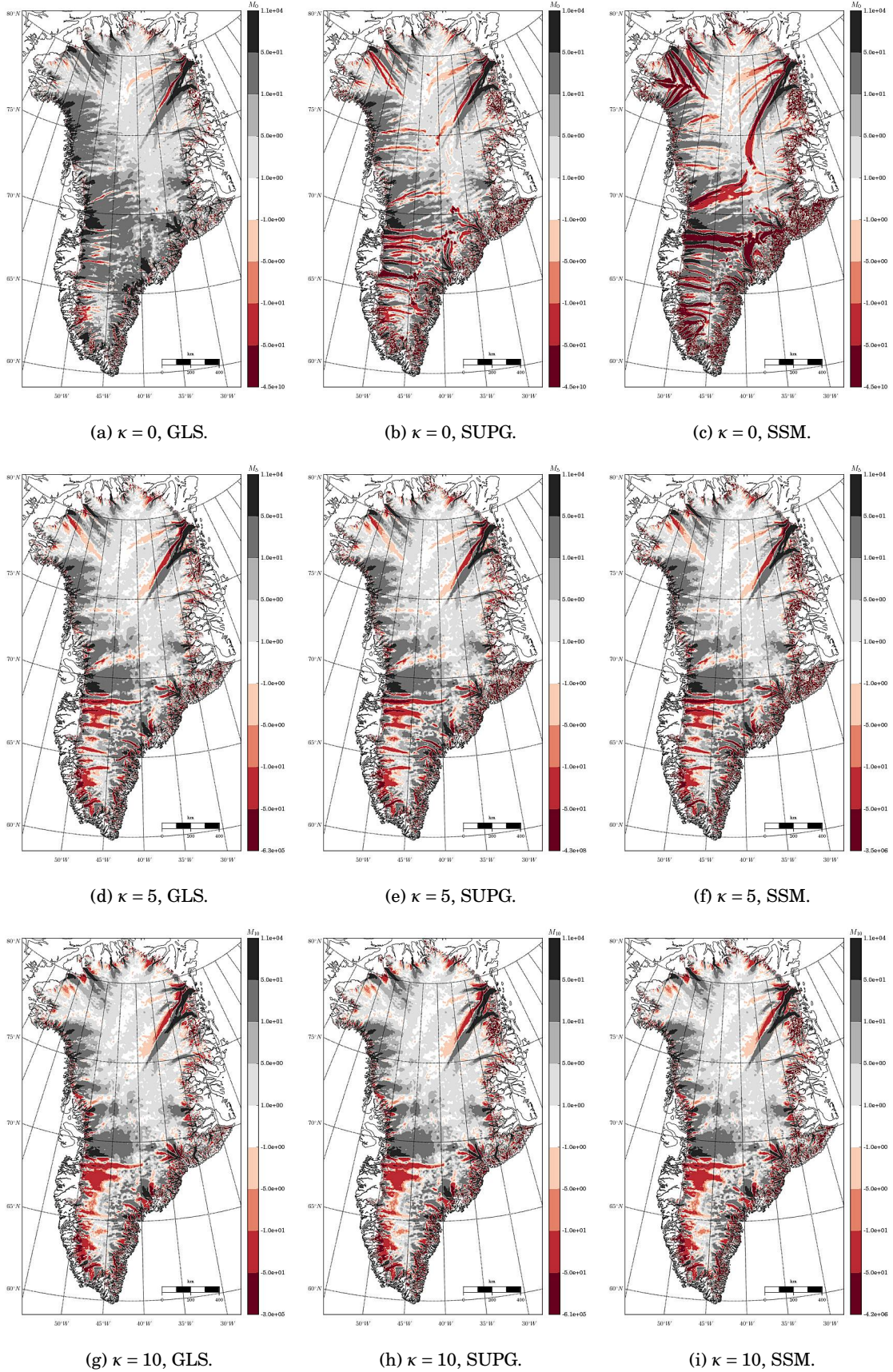


Figure 13.5: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} derived over Greenland with imposed direction of flow in the direction of surface velocity observations \mathbf{u}_{ob} , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23), streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24), or subgrid-scale-model (SSM) stabilization (13.25) in variational form (13.27).

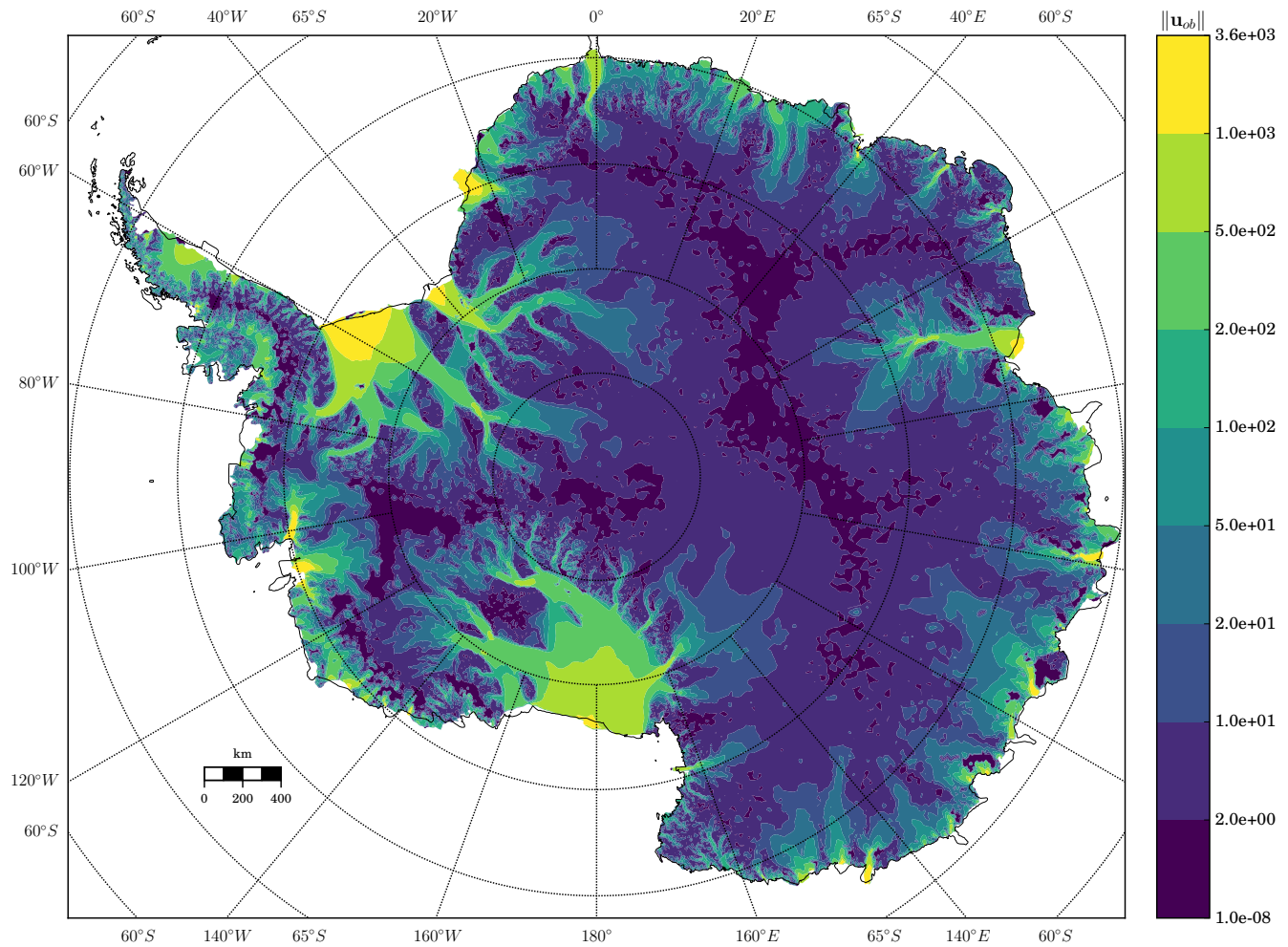


Figure 13.6: Surface velocity magnitude of Antarctica as provided by Rignot, Mouginot, and Scheuchl (2011).

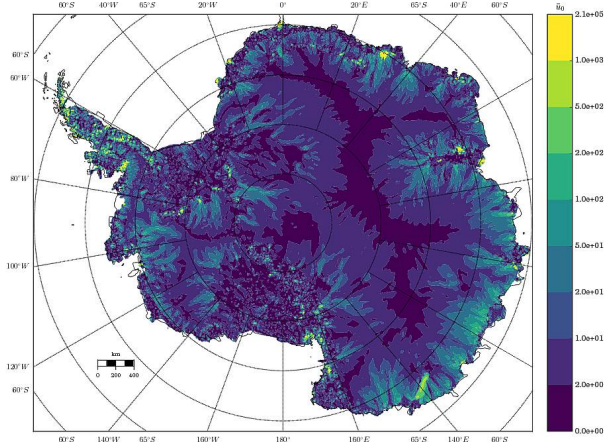
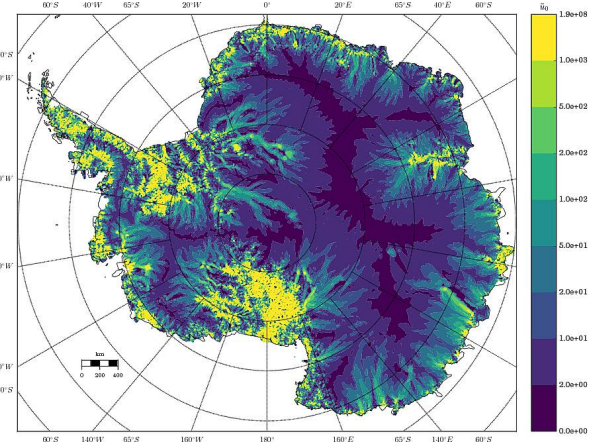
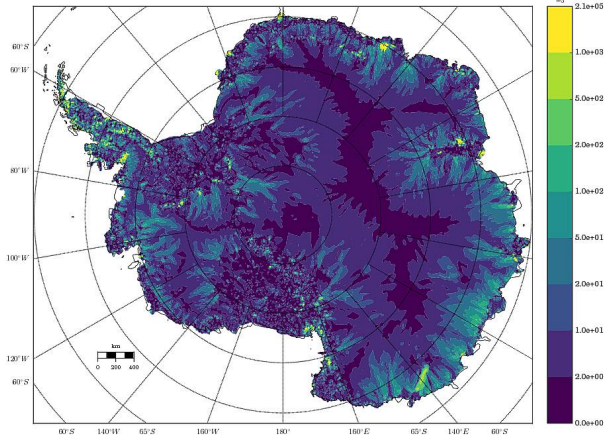
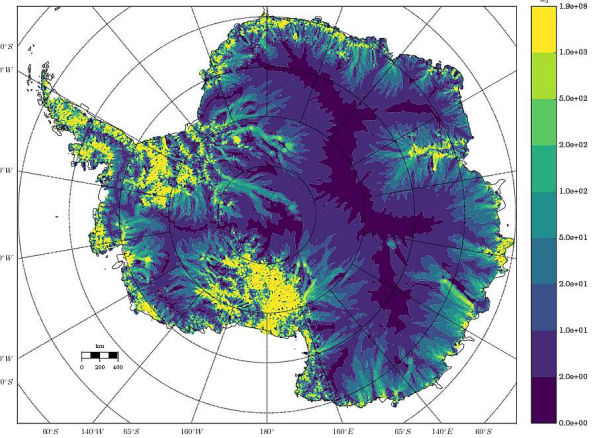
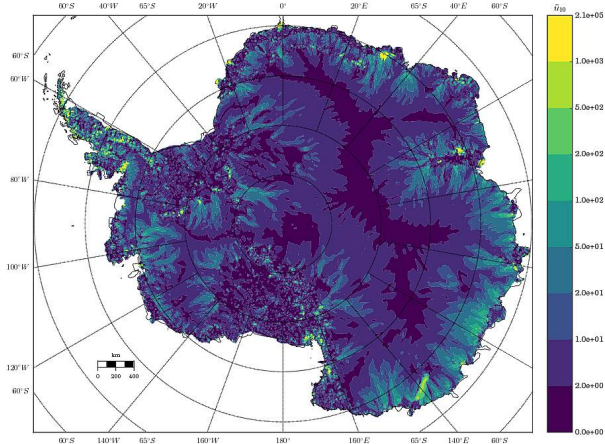
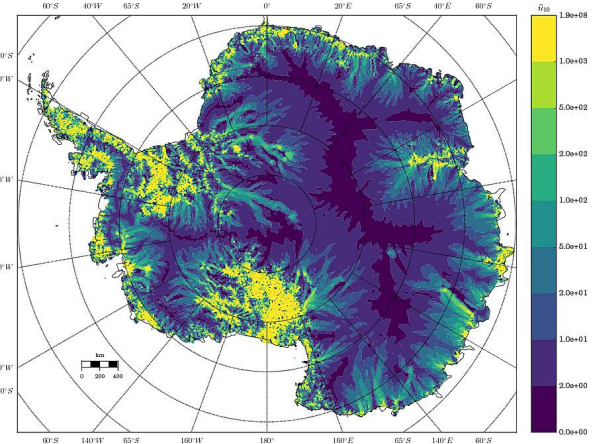
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.7: Balance velocity \bar{u} derived over Antarctica with imposed direction of flow down the surface gradient ∇S , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27). Results using subgrid-scale-model stabilization (13.25) (not shown) appeared more unstable than the (SUPG) method.

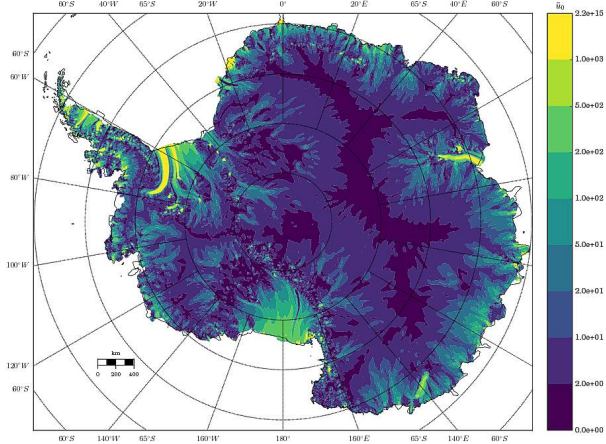
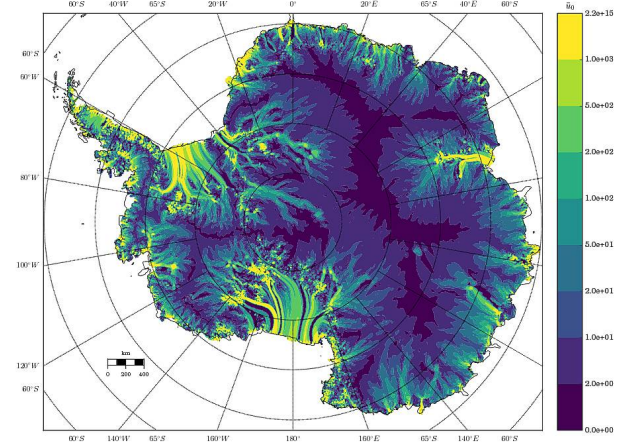
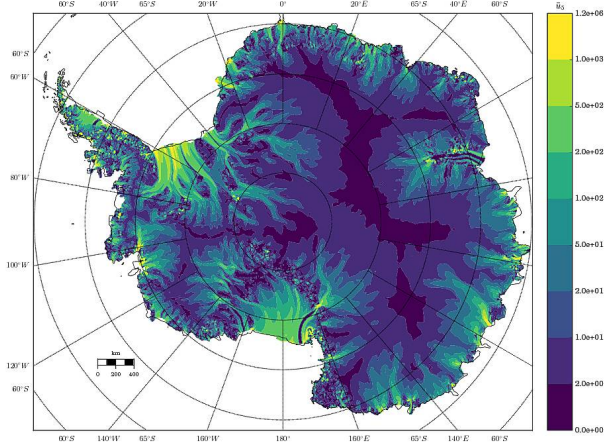
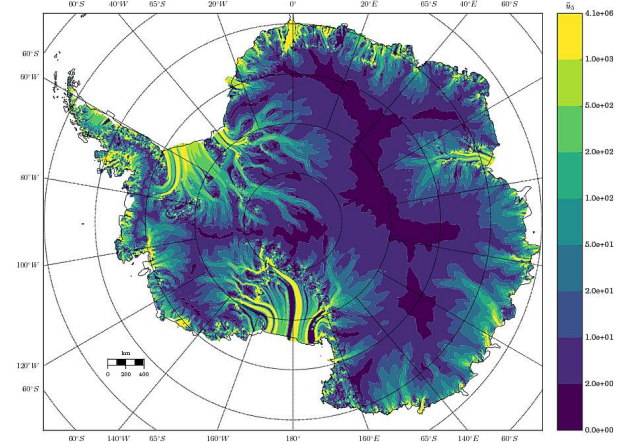
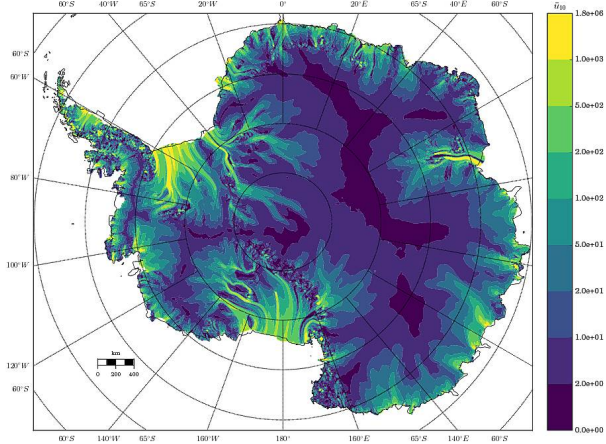
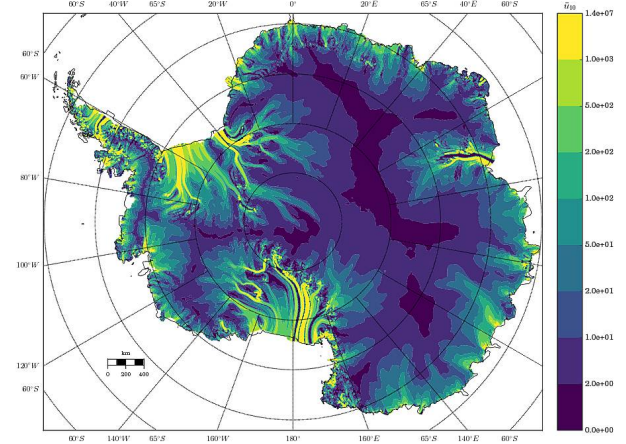
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.8: Balance velocity \bar{u} derived over Antarctica with imposed direction of flow down the surface gradient ∇S over grounded ice and in the direction of surface observations \mathbf{u}_{ob} over floating ice, where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27). Results using subgrid-scale-model stabilization (13.25) (not shown) appeared more unstable than the (SUPG) method.

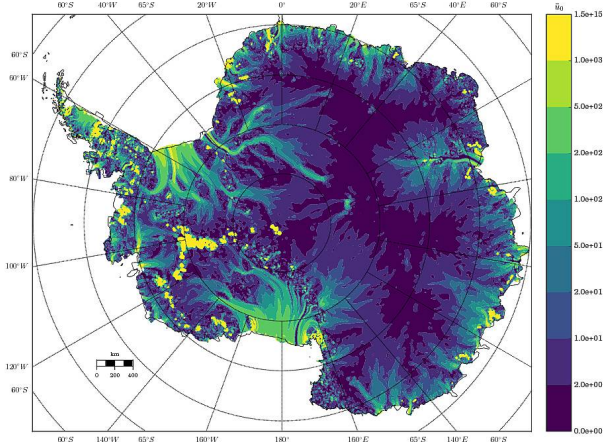
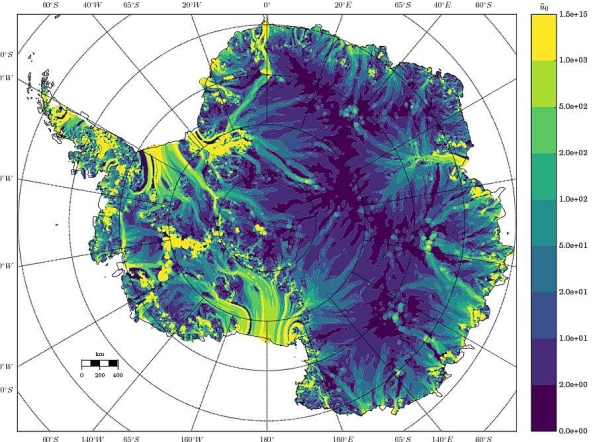
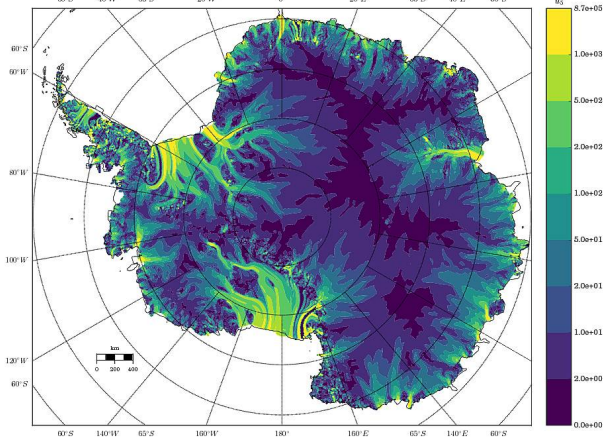
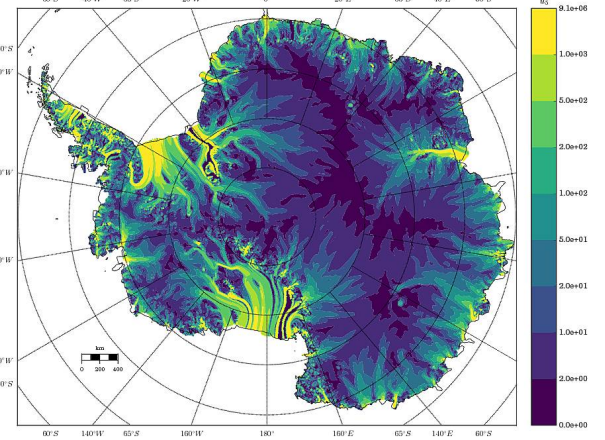
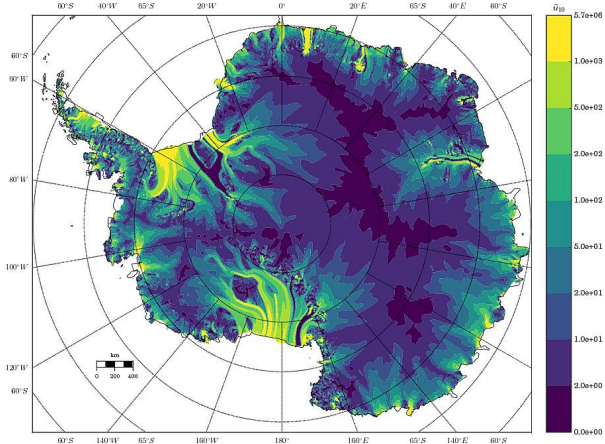
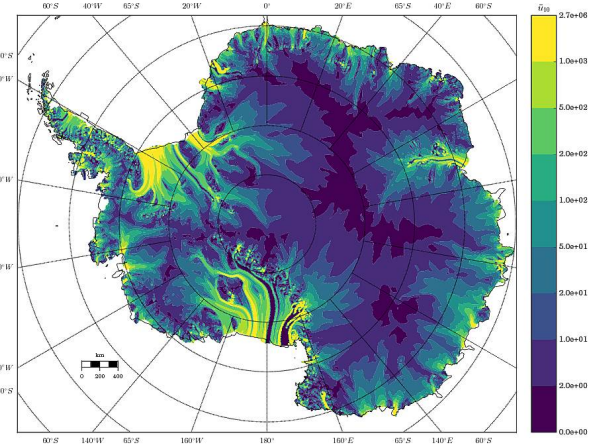
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.9: Balance velocity \bar{u} derived over Antarctica with imposed direction of flow in the direction of surface observations \mathbf{u}_{ob} , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27). Results using subgrid-scale-model stabilization (13.25) (not shown) appeared more unstable than the (SUPG) method.

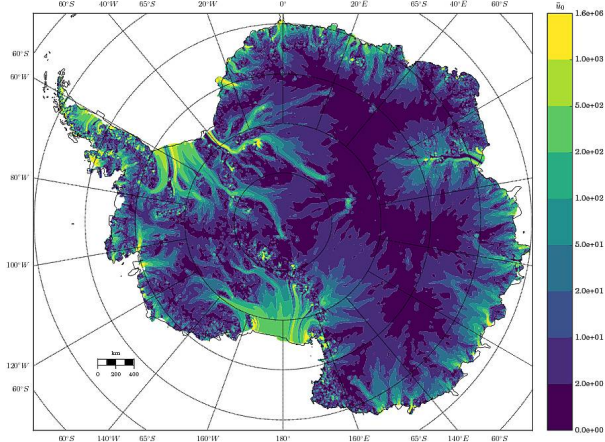
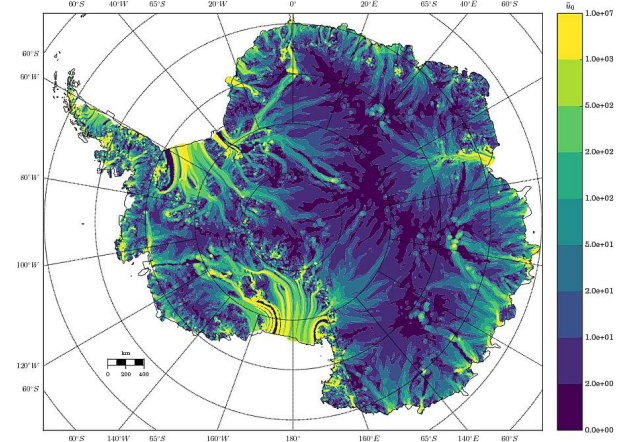
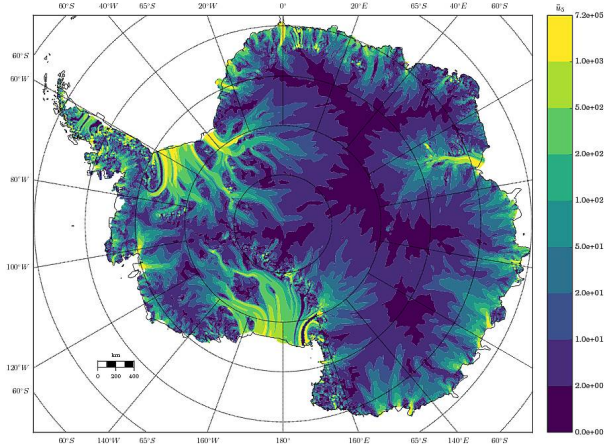
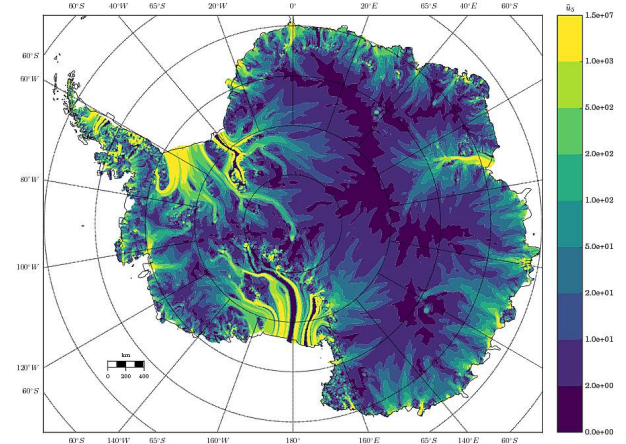
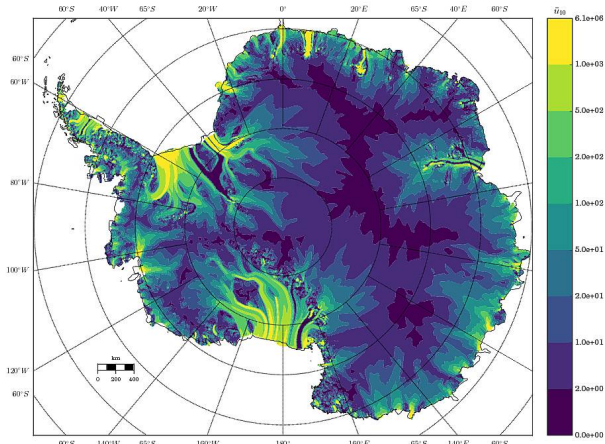
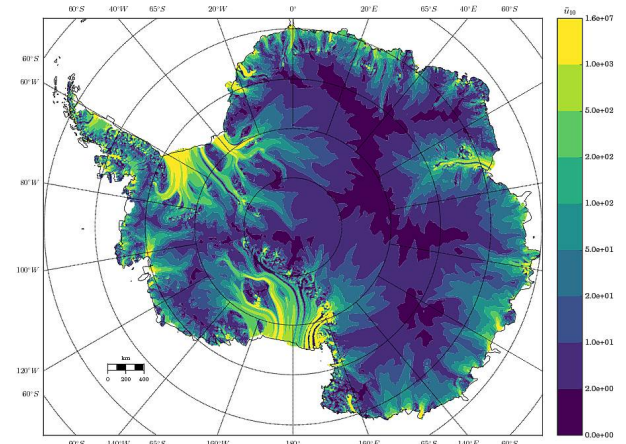
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.10: Balance velocity \bar{u} derived over Antarctica with imposed direction of flow in the direction of surface observations \mathbf{u}_{ob} and down the surface gradient ∇S where \mathbf{u}_{ob} values are missing, where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27). Results using subgrid-scale-model stabilization (13.25) (not shown) appeared more unstable than the (SUPG) method.

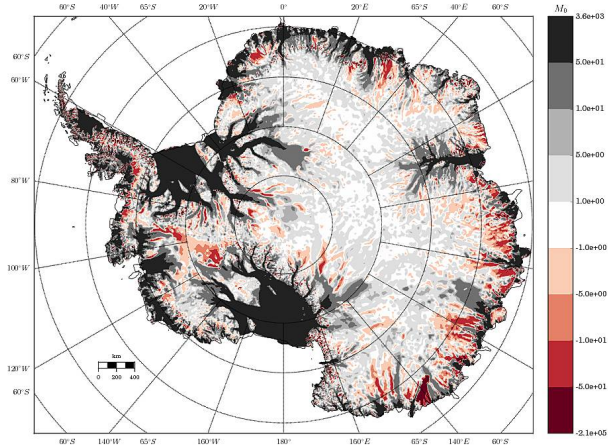
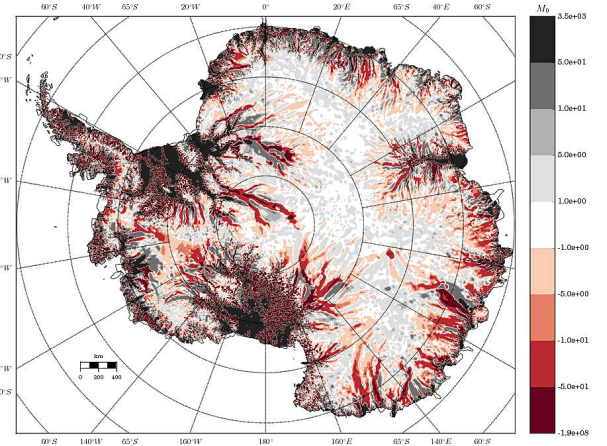
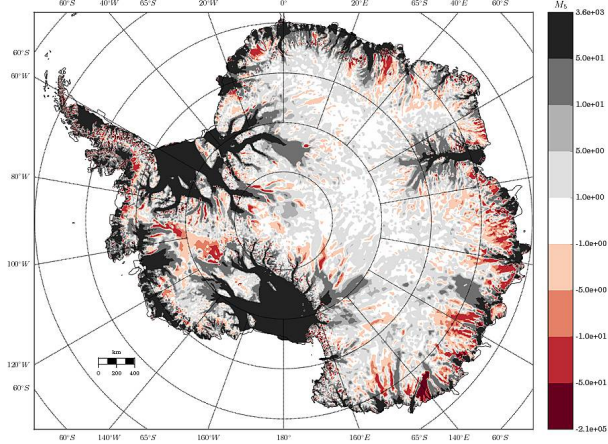
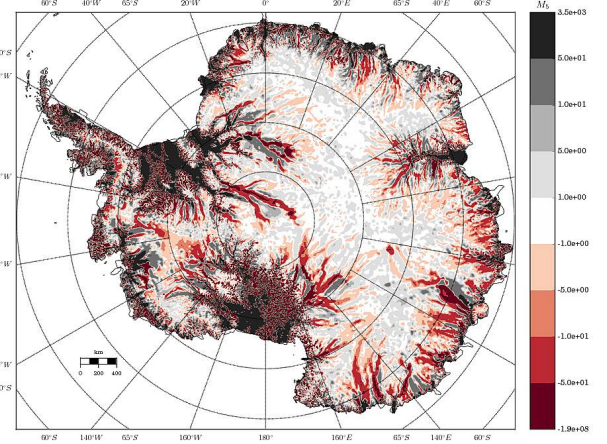
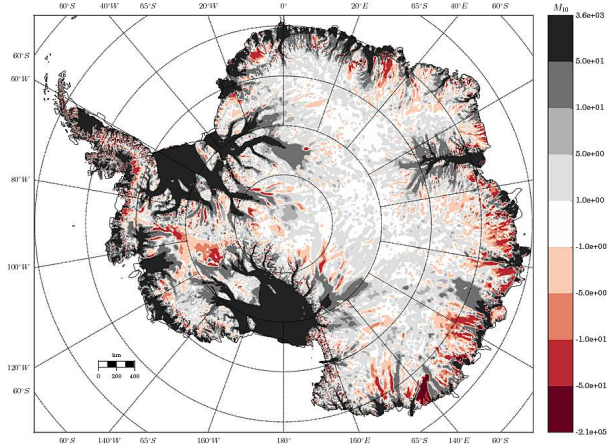
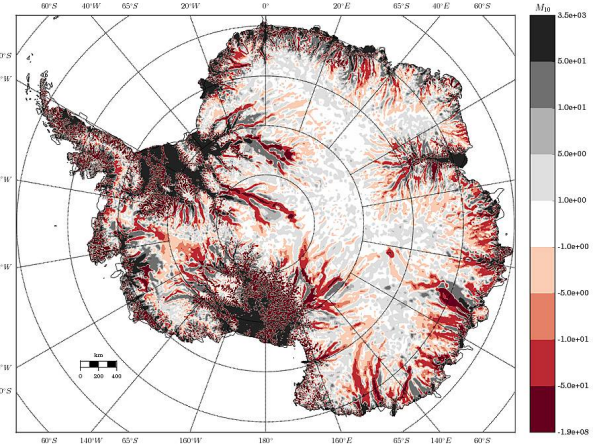
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.11: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} over Antarctica with imposed direction of flow down the surface gradient ∇S , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27).

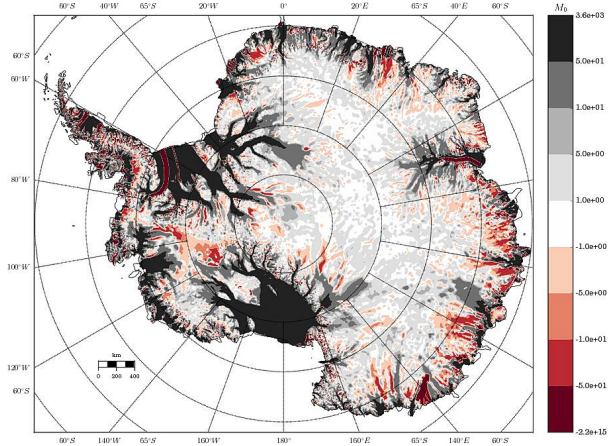
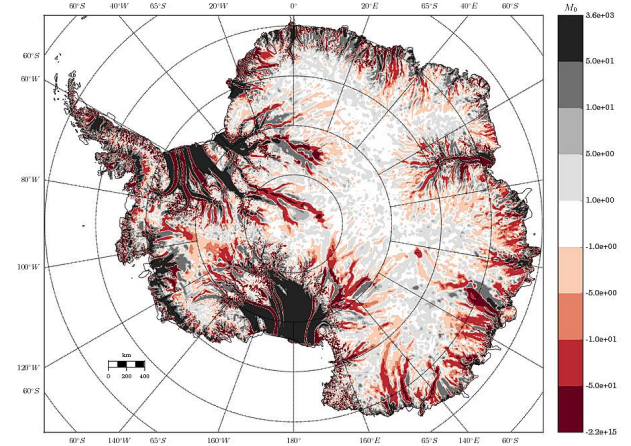
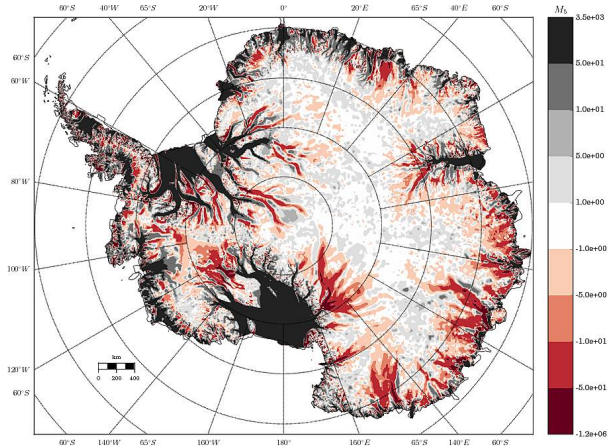
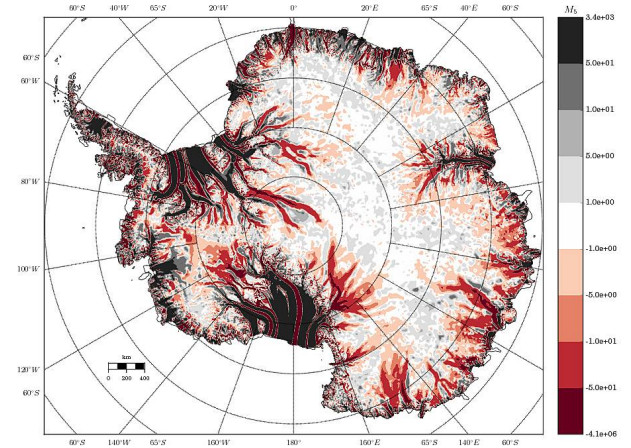
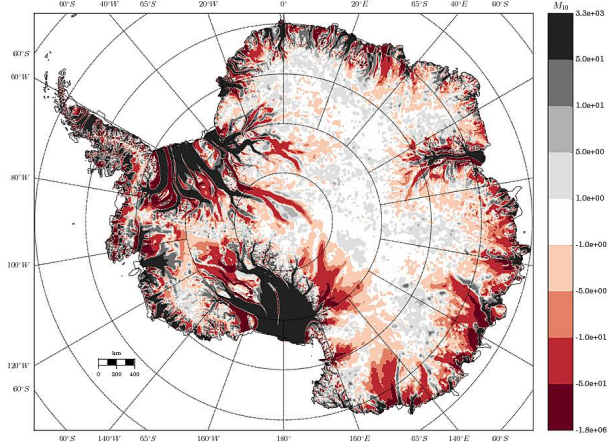
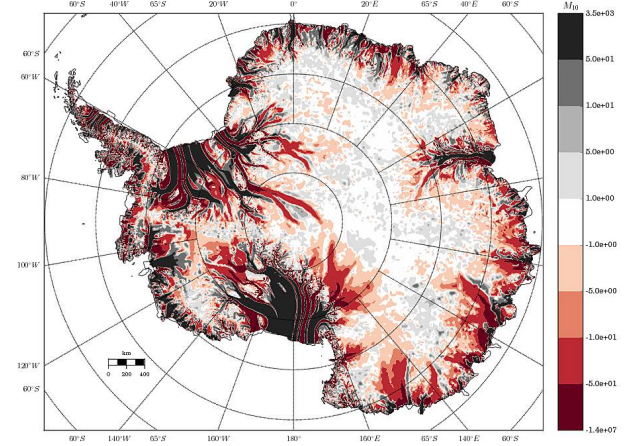
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.12: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} over Antarctica with imposed direction of flow down the surface gradient ∇S over grounded ice and in the direction of surface observations \mathbf{u}_{ob} over floating ice, where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27).

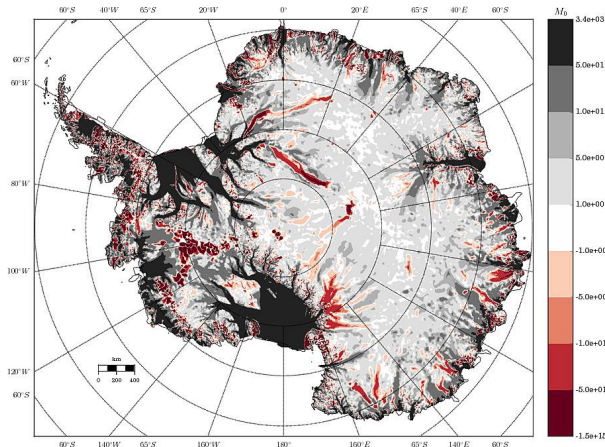
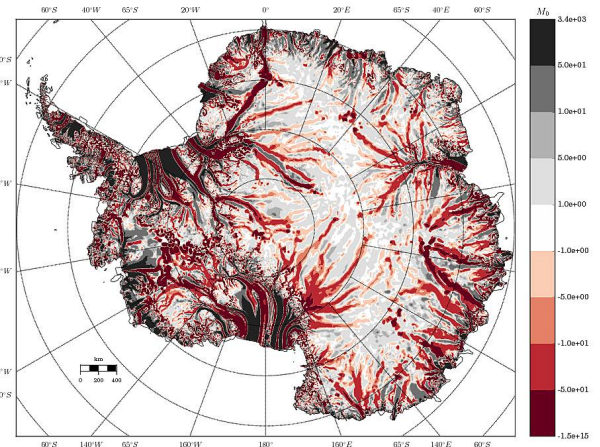
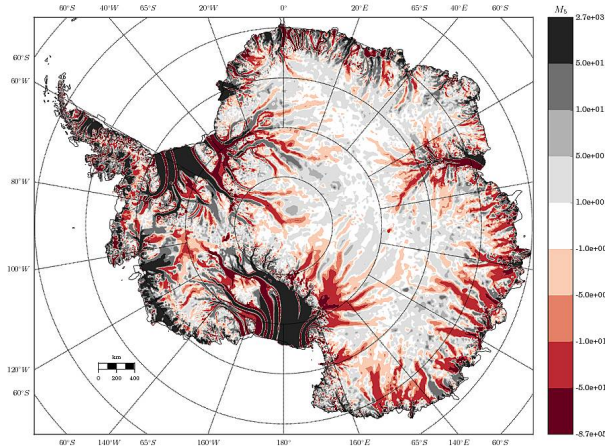
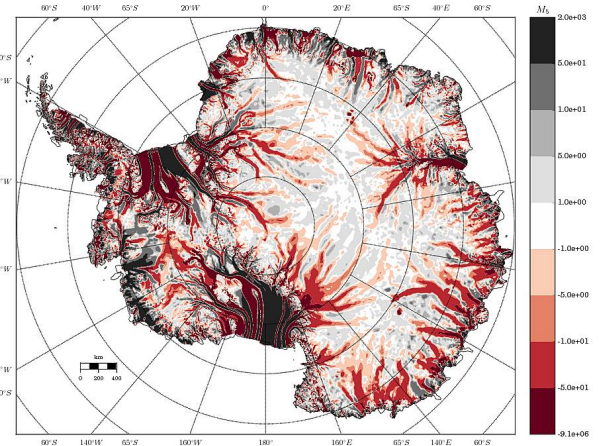
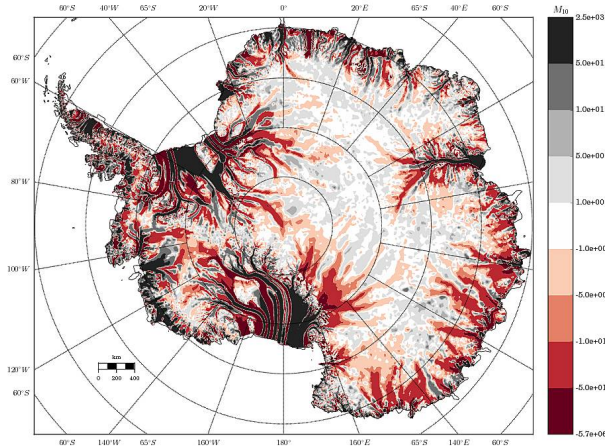
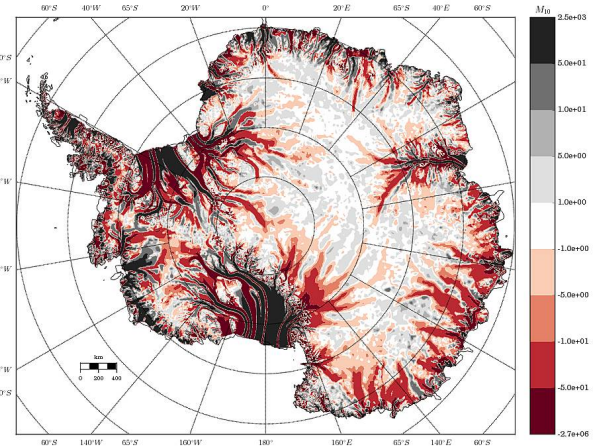
(a) $\kappa = 0$, GLS.(b) $\kappa = 0$, SUPG.(c) $\kappa = 5$, GLS.(d) $\kappa = 5$, SUPG.(e) $\kappa = 10$, GLS.(f) $\kappa = 10$, SUPG.

Figure 13.13: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} over Antarctica with imposed direction of flow in the direction of surface observations \mathbf{u}_{ob} , where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27).

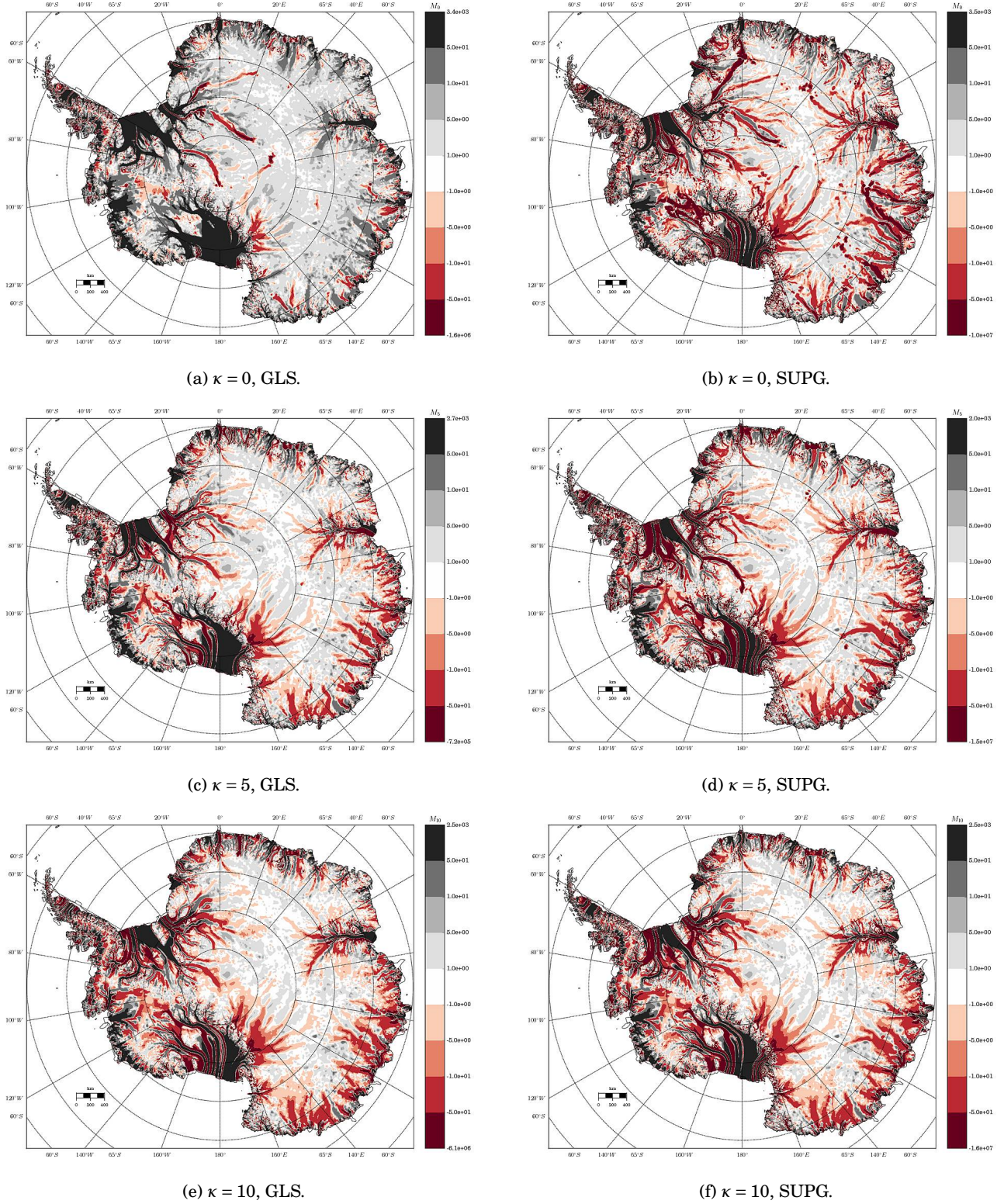


Figure 13.14: Difference $\|\mathbf{u}_{ob}\| - \bar{u}$ between balance velocity \bar{u} and the magnitude of the observed surface velocity \mathbf{u}_{ob} over Antarctica with imposed direction of flow in the direction of surface observations \mathbf{u}_{ob} and down the surface gradient ∇S where \mathbf{u}_{ob} values are missing, where smoothing radius κ varies as indicated. The columns vary according to stabilization used; either Galerkin/least-squares (GLS) stabilization (13.23) or streamline-upwind/Petrov-Galerkin (SUPG) stabilization (13.24) in variational form (13.27). Results using subgrid-scale-model stabilization (13.25) (not shown) appeared more unstable than the (SUPG) method.

Chapter 14

Stress balance

In this chapter we decompose the state of stress with an ice-sheet into along-flow and across-flow components

$$\sigma_n = \sigma \cdot \hat{\mathbf{u}}_n, \quad \sigma_t = \sigma \cdot \hat{\mathbf{u}}_t,$$

where the unit-vectors $\hat{\mathbf{u}}_n$ and $\hat{\mathbf{u}}_t$ point in the direction of and tangential to the flow, respectively:

$$\hat{\mathbf{u}}_n = \frac{[u \ v \ 0]^T}{\|\mathbf{u}_h\|}, \quad \text{and} \quad \hat{\mathbf{u}}_t = \frac{[v \ -u \ 0]^T}{\|\mathbf{u}_h\|}, \quad (14.1)$$

where $\mathbf{f}_h = [f_x \ f_y]^T$ denotes the horizontal components of the vector \mathbf{f} .

14.1 Membrane stress

Let coordinates (i, j, z) be the transformed velocity-coordinate system in the direction of flow, tangential to flow, and in the positive vertical direction, respectively. The unit vectors of this rotated coordinate system are

$$\hat{\mathbf{u}} = [\hat{u} \ \hat{v} \ 0]^T \quad \hat{\mathbf{v}} = [\hat{v} \ -\hat{u} \ 0]^T \quad \hat{\mathbf{w}} = [0 \ 0 \ \hat{w}]^T, \quad (14.2)$$

with normalized velocity vector components

$$\hat{u} = \frac{u}{\|\mathbf{u}_h\|}, \quad \hat{v} = \frac{v}{\|\mathbf{u}_h\|}, \quad \hat{w} = 1. \quad (14.3)$$

The partial derivatives in this new coordinate system can be evaluated using the directional derivatives

$$\begin{aligned} \frac{\partial f}{\partial i} &= \nabla_{\hat{\mathbf{u}}} f = \nabla f \cdot \hat{\mathbf{u}} = \frac{\partial f}{\partial x} \hat{u} + \frac{\partial f}{\partial y} \hat{v} \\ \frac{\partial f}{\partial j} &= \nabla_{\hat{\mathbf{v}}} f = \nabla f \cdot \hat{\mathbf{v}} = \frac{\partial f}{\partial x} \hat{v} - \frac{\partial f}{\partial y} \hat{u} \\ \frac{\partial f}{\partial z} &= \nabla_{\hat{\mathbf{w}}} f = \nabla f \cdot \hat{\mathbf{w}} = \frac{\partial f}{\partial z}, \end{aligned}$$

and the z -rotated gradient operator

$$\nabla_r \equiv \left[\frac{\partial}{\partial i} \ \frac{\partial}{\partial j} \ \frac{\partial}{\partial z} \right]^T. \quad (14.4)$$

Next, let ϕ be the signed angle in radians between the x -axis and the horizontal velocity vector $\mathbf{u}_h = [u \ v]^T$. The rotation

matrix about the z -axis used to transform any vector in the x, y, z coordinate system to the i, j, z coordinate system is

$$R_z = \begin{bmatrix} \cos(\phi) & -\sin(\phi) & 0 \\ \sin(\phi) & \cos(\phi) & 0 \\ 0 & 0 & 1 \end{bmatrix},$$

and the ϕ -rotation of the rank-two Cauchy-stress tensor σ about the z -axis is

$$\sigma_r = R_z \cdot \sigma \cdot R_z^T = \begin{bmatrix} \sigma_{ii} & \sigma_{ij} & \sigma_{iz} \\ \sigma_{ji} & \sigma_{jj} & \sigma_{jz} \\ \sigma_{zi} & \sigma_{zj} & \sigma_{zz} \end{bmatrix}. \quad (14.5)$$

Returning to momentum balance (8.1), the ϕ -rotated momentum balance is

$$-\nabla_r \cdot \sigma_r = \mathbf{f}, \quad (14.6)$$

leading to an expansion similar to (9.6),

$$\begin{aligned} \frac{\partial \sigma_{ii}}{\partial i} + \frac{\partial \sigma_{ij}}{\partial j} + \frac{\partial \sigma_{iz}}{\partial z} &= 0 \\ \frac{\partial \sigma_{ji}}{\partial i} + \frac{\partial \sigma_{jj}}{\partial j} + \frac{\partial \sigma_{jz}}{\partial z} &= 0 \\ \frac{\partial \sigma_{zi}}{\partial i} + \frac{\partial \sigma_{zj}}{\partial j} + \frac{\partial \sigma_{zz}}{\partial z} &= \rho g. \end{aligned}$$

Next, rotated momentum-balance (14.6) is integrated vertically,

$$-\int_B^S \nabla_r \cdot \sigma_r \, dz = \int_B^S \mathbf{f} \, dz. \quad (14.7)$$

Similar to the derivation of vertically-integrated mass-balance (13.11), Leibniz's rule is applied to the above (Appendix B), resulting in the vertically-integrated stress-balance

$$-\nabla_r \cdot \left(\int_B^S \sigma_r \, dz \right) + \sigma_r|_S \cdot \nabla_r S - \sigma_r|_B \cdot \nabla_r B = \int_B^S \mathbf{f} \, dz. \quad (14.8)$$

It is also of interest to examine the state of stress without the contribution of the mean compressive stress $p = -\sigma_{kk}/3$, and thus perform a similar set of calculations as above to deviatoric stress-tensor τ in (8.4). The rotated-stress-deviator tensor is thus

$$\tau_r = R_z \cdot \tau \cdot R_z^T = \begin{bmatrix} \tau_{ii} & \tau_{ij} & \tau_{iz} \\ \tau_{ji} & \tau_{jj} & \tau_{jz} \\ \tau_{zi} & \tau_{zj} & \tau_{zz} \end{bmatrix}, \quad (14.9)$$

such that stress tensor (14.5) may be decomposed using (8.4) into

$$\sigma_r = \tau_r - pI = \begin{bmatrix} \tau_{ii} & \tau_{ij} & \tau_{iz} \\ \tau_{ji} & \tau_{jj} & \tau_{jz} \\ \tau_{zi} & \tau_{zj} & \tau_{zz} \end{bmatrix} - p \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

Therefore, using the fact that $\nabla_r \cdot (pI) = \nabla_r p$, vertically-integrated stress-balance (14.8) may also be written

$$-\nabla_r \cdot \left(\int_B^S \tau_r dz \right) + \tau_r|_S \cdot \nabla_r S - \tau_r|_B \cdot \nabla_r B = \int_B^S (\mathbf{f} - \nabla_r p) dz. \quad (14.10)$$

Finally, the terms of the vertically-integrated deviatoric-stress tensor $\int_z \tau_r dz = N$ contains individual components

$$N = \begin{bmatrix} N_{ii} & N_{ij} & N_{iz} \\ N_{ji} & N_{jj} & N_{jz} \\ N_{zi} & N_{zj} & N_{zz} \end{bmatrix} = \begin{bmatrix} \int_z \tau_{ii} & \int_z \tau_{ij} & \int_z \tau_{iz} \\ \int_z \tau_{ji} & \int_z \tau_{jj} & \int_z \tau_{jz} \\ \int_z \tau_{zi} & \int_z \tau_{zj} & \int_z \tau_{zz} \end{bmatrix}, \quad (14.11)$$

and are referred to as *membrane stresses* (Greve and Blatter, 2009).

14.2 Membrane stress balance

Re-writing stress balance (14.7) using (14.10), we have an equivalent form of (14.10), the *membrane stress balance*

$$-M\mathbf{1} = \mathbf{f}_{\text{int}}, \quad (14.12)$$

where $\mathbf{1} = [1 \ 1 \ 1]^T$ is the rank-one tensor of ones, $\mathbf{f}_{\text{int}} = \int_z (\mathbf{f} - \nabla_r p) dz$ is the right-hand side of (14.10), and the tensor M is defined as

$$M = \begin{bmatrix} M_{ii} & M_{ij} & M_{iz} \\ M_{ji} & M_{jj} & M_{jz} \\ M_{zi} & M_{zj} & M_{zz} \end{bmatrix} = \begin{bmatrix} \int_z \frac{\partial \tau_{ii}}{\partial i} & \int_z \frac{\partial \tau_{ij}}{\partial j} & \int_z \frac{\partial \tau_{iz}}{\partial z} \\ \int_z \frac{\partial \tau_{ji}}{\partial i} & \int_z \frac{\partial \tau_{jj}}{\partial j} & \int_z \frac{\partial \tau_{jz}}{\partial z} \\ \int_z \frac{\partial \tau_{zi}}{\partial i} & \int_z \frac{\partial \tau_{zj}}{\partial j} & \int_z \frac{\partial \tau_{zz}}{\partial z} \end{bmatrix}. \quad (14.13)$$

Applying Leibniz's rule to the i - and j -derivative terms, and the first fundamental theorem of calculus to the z -derivative terms,

$$\left. \begin{aligned} M_{ii} &= \frac{\partial}{\partial i} N_{ii} + \tau_{ii}(S) \frac{\partial S}{\partial i} - \tau_{ii}(B) \frac{\partial B}{\partial i} \\ M_{ij} &= \frac{\partial}{\partial j} N_{ij} + \tau_{ij}(S) \frac{\partial S}{\partial j} - \tau_{ij}(B) \frac{\partial B}{\partial j} \\ M_{iz} &= \tau_{iz}(S) - \tau_{iz}(B) \\ M_{ji} &= \frac{\partial}{\partial i} N_{ji} + \tau_{ji}(S) \frac{\partial S}{\partial i} - \tau_{ji}(B) \frac{\partial B}{\partial i} \\ M_{jj} &= \frac{\partial}{\partial j} N_{jj} + \tau_{jj}(S) \frac{\partial S}{\partial j} - \tau_{jj}(B) \frac{\partial B}{\partial j} \\ M_{jz} &= \tau_{jz}(S) - \tau_{jz}(B) \\ M_{zi} &= \frac{\partial}{\partial i} N_{zi} + \tau_{zi}(S) \frac{\partial S}{\partial i} - \tau_{zi}(B) \frac{\partial B}{\partial i} \\ M_{zj} &= \frac{\partial}{\partial j} N_{zj} + \tau_{zj}(S) \frac{\partial S}{\partial j} - \tau_{zj}(B) \frac{\partial B}{\partial j} \\ M_{zz} &= \tau_{zz}(S) - \tau_{zz}(B) \end{aligned} \right\}. \quad (14.14)$$

Provided that the elements of tensors σ and τ have been populated with values obtained by solving one of the three-dimensional momentum-balance formulations of Chapter 9, the components of membrane-stress tensor (14.11) may be calculated using numerical integration (see §2.3 for an analogous problem in one dimension). These stresses, once derived, may then be used to calculate the individual stress terms of membrane-stress balance (14.12) given by (14.14).

Finally, note that the last row of first-order strain-rate tensor (9.20) has been eliminated. While it is surely possible to employ full-Cauchy stress-deviator tensor definition (8.4) to evaluate membrane-stress tensor (14.11) using a velocity field computed from first-order momentum balance (9.32), it is more instructive to examine the state of stress for this model from the point of view of its mathematical formulation. Therefore, the first-order strain-rate tensor, defined as

$$\tilde{\epsilon} = \begin{bmatrix} \frac{\partial u}{\partial x} & \frac{1}{2} \left(\frac{\partial u}{\partial y} + \frac{\partial v}{\partial x} \right) & \frac{1}{2} \frac{\partial u}{\partial z} \\ \frac{1}{2} \left(\frac{\partial v}{\partial x} + \frac{\partial u}{\partial y} \right) & \frac{\partial v}{\partial y} & \frac{1}{2} \frac{\partial v}{\partial z} \\ \frac{1}{2} \frac{\partial u}{\partial z} & \frac{1}{2} \frac{\partial v}{\partial z} & - \left(\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} \right) \end{bmatrix}, \quad (14.15)$$

and first-order stress-deviator tensor

$$\tau_{\text{BP}} = 2\eta_{\text{BP}} \tilde{\epsilon}, \quad (14.16)$$

derived from the simplifications described in §9.2.2 are used to evaluate the balance of stress associated with this model.

The CSLVR implementation of this problem is shown in Code Listing 14.1.

Code Listing 14.1: CSLVR source code for the StressBalance class.

```
from csivr.physics import Physics
from csivr.io import print_text, print_min_max
from fenics import *
```

```
class StressBalance(Physics):
    def __init__(self, model, momentum):
        """
        """
        s = ">::: INITIALIZING STRESS-BALANCE PHYSICS :::"
        print_text(s, self.color())

        # calculate viscosity in model.eta :
        U = momentum.velocity()
        epsdot = momentum.effective_strain_rate(U) + model.eps_reg
        model.calc_eta(epsdot)

        # stress tensor :
        tau = momentum.deviatoric_stress_tensor(U, model.eta)

        # rotate about the z-axis :
        rad_xy = model.get_xy_velocity_angle(U)
        Rz = model.z_rotation_matrix(rad_xy)
        tau_r = model.rotate_tensor(tau, Rz)

        # get surface stress :
        tau_ii_S = model.vert_extrude(tau_r[0,0], d='down', Q=model.Q_non_periodic)
        tau_ij_S = model.vert_extrude(tau_r[0,1], d='down', Q=model.Q_non_periodic)
        tau_iz_S = model.vert_extrude(tau_r[0,2], d='down', Q=model.Q_non_periodic)
        tau_ji_S = model.vert_extrude(tau_r[1,0], d='down', Q=model.Q_non_periodic)
        tau_jj_S = model.vert_extrude(tau_r[1,1], d='down', Q=model.Q_non_periodic)
        tau_jz_S = model.vert_extrude(tau_r[1,2], d='down', Q=model.Q_non_periodic)
        tau_zi_S = model.vert_extrude(tau_r[2,0], d='down', Q=model.Q_non_periodic)
        tau_zj_S = model.vert_extrude(tau_r[2,1], d='down', Q=model.Q_non_periodic)
        tau_zz_S = model.vert_extrude(tau_r[2,2], d='down', Q=model.Q_non_periodic)

        # get basal stress :
        tau_ii_B = model.vert_extrude(tau_r[0,0], d='up', Q=model.Q_non_periodic)
        tau_ij_B = model.vert_extrude(tau_r[0,1], d='up', Q=model.Q_non_periodic)
        tau_iz_B = model.vert_extrude(tau_r[0,2], d='up', Q=model.Q_non_periodic)
        tau_ji_B = model.vert_extrude(tau_r[1,0], d='up', Q=model.Q_non_periodic)
        tau_jj_B = model.vert_extrude(tau_r[1,1], d='up', Q=model.Q_non_periodic)
        tau_jz_B = model.vert_extrude(tau_r[1,2], d='up', Q=model.Q_non_periodic)
        tau_zi_B = model.vert_extrude(tau_r[2,0], d='up', Q=model.Q_non_periodic)
        tau_zj_B = model.vert_extrude(tau_r[2,1], d='up', Q=model.Q_non_periodic)
        tau_zz_B = model.vert_extrude(tau_r[2,2], d='up', Q=model.Q_non_periodic)

        # vertically integrate deviatoric stress (membrane stress) :
        t_ii = model.vert_integrate(tau_r[0,0], d='up', Q=model.Q_non_periodic)
        t_ij = model.vert_integrate(tau_r[0,1], d='up', Q=model.Q_non_periodic)
        t_iz = model.vert_integrate(tau_r[0,2], d='up', Q=model.Q_non_periodic)
        t_ji = model.vert_integrate(tau_r[1,0], d='up', Q=model.Q_non_periodic)
        t_jj = model.vert_integrate(tau_r[1,1], d='up', Q=model.Q_non_periodic)
        t_jz = model.vert_integrate(tau_r[1,2], d='up', Q=model.Q_non_periodic)
        t_zi = model.vert_integrate(tau_r[2,0], d='up', Q=model.Q_non_periodic)
        t_zj = model.vert_integrate(tau_r[2,1], d='up', Q=model.Q_non_periodic)
        t_zz = model.vert_integrate(tau_r[2,2], d='up', Q=model.Q_non_periodic)
```



```

# extrude the integral down the vertical :
M_ii = model.vert_extrude(t_ii, d='down', Q=model.Q_non_periodic)
M_ij = model.vert_extrude(t_ij, d='down', Q=model.Q_non_periodic)
M_iz = model.vert_extrude(t_iz, d='down', Q=model.Q_non_periodic)
M_ji = model.vert_extrude(t_ji, d='down', Q=model.Q_non_periodic)
M_jj = model.vert_extrude(t_jj, d='down', Q=model.Q_non_periodic)
M_jz = model.vert_extrude(t_jz, d='down', Q=model.Q_non_periodic)
M_zi = model.vert_extrude(t_zi, d='down', Q=model.Q_non_periodic)
M_zj = model.vert_extrude(t_zj, d='down', Q=model.Q_non_periodic)
M_zz = model.vert_extrude(t_zz, d='down', Q=model.Q_non_periodic)

# save the membrane stresses :
model.init_M_ii(M_ii, cls=self)
model.init_M_ij(M_ij, cls=self)
model.init_M_iz(M_iz, cls=self)
model.init_M_ji(M_ji, cls=self)
model.init_M_jj(M_jj, cls=self)
model.init_M_jz(M_jz, cls=self)
model.init_M_zi(M_zi, cls=self)
model.init_M_zj(M_zj, cls=self)
model.init_M_zz(M_zz, cls=self)

# get the components of horizontal velocity :
u,v,w = U.split(True)
U = as_vector([u, v])
U_hat = model.normalize_vector(U)
U_n = as_vector([U_hat[0], -U_hat[1], 0.0])
U_t = as_vector([U_hat[1], U_hat[0], 0.0])
S = model.S
B = model.B

# directional derivative in direction of flow :
def d_di(u): return dot(grad(u), U_n)

# directional derivative in direction across flow :
def d_dj(u): return dot(grad(u), U_t)

# form components :
phi = TestFunction(model.Q_non_periodic)
dtau = TrialFunction(model.Q_non_periodic)

# mass matrix :
self.M = assemble(phi*dtau*dx)

# integrated stress-balance using Leibniz Theorem :
self.M_ii = (d_di(M_ii) + tau_ii.S*d_di(S) - tau_ii.B*d_di(B)) * phi * dx
self.M_ij = (d_dj(M_ij) + tau_ij.S*d_dj(S) - tau_ij.B*d_dj(B)) * phi * dx
self.M_iz = (tau_iz.S - tau_iz.B) * phi * dx
self.M_ji = (d_di(M_ji) + tau_ji.S*d_di(S) - tau_ji.B*d_di(B)) * phi * dx
self.M_jj = (d_dj(M_jj) + tau_jj.S*d_dj(S) - tau_jj.B*d_dj(B)) * phi * dx
self.M_jz = (tau_jz.S - tau_jz.B) * phi * dx
self.M_zi = (d_di(M_zi) + tau_zi.S*d_di(S) - tau_zi.B*d_di(B)) * phi * dx
self.M_zj = (d_dj(M_zj) + tau_zj.S*d_dj(S) - tau_zj.B*d_dj(B)) * phi * dx
self.M_zz = (tau_zz.S - tau_zz.B) * phi * dx

def solve(self):
    """
    s = " :: solving 'Stress_Balance' :: "
    print_text(s, self.color())

    model = self.model

    # solve the linear system :
    solve(self.M, model.M_ii.vector(), assemble(self.M_ii))
    print_min_max(model.M_ii, 'M_ii')
    solve(self.M, model.M_ij.vector(), assemble(self.M_ij))
    print_min_max(model.M_ij, 'M_ij')
    solve(self.M, model.M_iz.vector(), assemble(self.M_iz))
    print_min_max(model.M_iz, 'M_iz')
    solve(self.M, model.M_ji.vector(), assemble(self.M_ji))
    print_min_max(model.M_ji, 'M_ji')
    solve(self.M, model.M_jj.vector(), assemble(self.M_jj))
    print_min_max(model.M_jj, 'M_jj')
    solve(self.M, model.M_jz.vector(), assemble(self.M_jz))
    print_min_max(model.M_jz, 'M_jz')
    solve(self.M, model.M_zi.vector(), assemble(self.M_zi))
    print_min_max(model.M_zi, 'M_zi')
    solve(self.M, model.M_zj.vector(), assemble(self.M_zj))
    print_min_max(model.M_zj, 'M_zj')
    solve(self.M, model.M_zz.vector(), assemble(self.M_zz))
    print_min_max(model.M_zz, 'M_zz')

```

14.3 ISMIP-HOM test simulation

In this section we revisit one of the higher-wavelength ISMIP-HOM experiment presented in §9.7 in order to examine the distributions of stress for each of the momentum models defined in §9.1, §9.2, and §9.4. Once again, this test is defined over the domain $\Omega \in [0, \ell] \times [0, \ell] \times [B, S] \subset \mathbb{R}^3$ with a $k_x \times k_y \times k_z$ element discretization, and specifies the use of a surface height with uniform slope $\|\nabla S\| = a$

$$S(x) = -x \tan(a),$$

and the sinusoidally-varying basal topography

$$B(x, y) = S(x) - \bar{B} + b \sin\left(\frac{2\pi}{\ell} x\right) \sin\left(\frac{2\pi}{\ell} y\right),$$

with average basal depth \bar{B} , and basal height amplitude b (Figure 9.1). To enforce continuity, the periodic \mathbf{u}, p boundary

conditions

$$\begin{aligned} \mathbf{u}(0, 0) &= \mathbf{u}(\ell, \ell) & p(0, 0) &= p(\ell, \ell) \\ \mathbf{u}(0, \ell) &= \mathbf{u}(\ell, 0) & p(0, \ell) &= p(\ell, 0) \\ \mathbf{u}(x, 0) &= \mathbf{u}(x, \ell) & p(x, 0) &= p(x, \ell) \\ \mathbf{u}(0, y) &= \mathbf{u}(\ell, y) & p(0, y) &= p(\ell, y) \end{aligned}$$

were applied. Lastly, the basal traction coefficient was set to $\beta = 1000$, having the effect of creating a no-slip boundary condition along the entire basal surface, while $A = 10^{-16}$ was used as an isothermal rate factor for viscosity η . Table 14.1 lists these and other coefficients used, and CSLVR script used to calculate the momentum- and stress-balance is shown in Code Listing 14.2.

The domain width $\ell = 8$ km was chosen for this analysis due to the fact that significant differences exist between solutions obtained by each of the higher-order models (see §9.7). Results indicate that the membrane-stress distributions associated with the reformulated-Stokes balance of §9.4 are more similar to that associated with the full-Stokes balance of §9.1 than the first-order model of §9.2 (Figures 14.1, 14.2, and 14.3). The most striking difference between the reformulated-Stokes and full-Stokes balance appears to be the z -normal stress N_{zz} involving the z -derivative of vertical velocity w ; the reformulated-Stokes solution is approximately one order of magnitude larger than the full-Stokes solution for this term (Figures 14.2k and 14.1k). It is also interesting to note that the oscillations in the y -direction of lateral-shearing term $N_{ij} = N_{ji}$ corresponding to the reformulated-Stokes model (Figures 14.2d and 14.2f) is one wavelength larger than the full-Stokes model (Figures 14.1d and 14.1f).

The distribution and magnitude of the membrane-stress-balance results associated with the first-order model (Figure 14.6) are both remarkably different from the full-Stokes results (Figure 14.4). This model appears to over-estimate the contribution of along-flow normal stress M_{ii} (Figure 14.6c); lateral-shear M_{ij} and M_{ji} (Figures 14.6d and 14.6f); and vertical-shear M_{iz} (Figure 14.6e), while under-estimating the remaining terms.

Code Listing 14.2: CSLVR script that performs the stress-balance calculation for the ISMIP-HOM problem of §14.3.

```

from csivr import *

# constants used :
a = 0.5 * pi / 180
L = 8000

# create the mesh :
p1 = Point(0.0, 0.0, 0.0)
p2 = Point(L, L, 1)
mesh = BoxMesh(p1, p2, 15, 15, 5)

out_dir = './results/'

# stress balance requires the solution to a 3D model, in this case periodic :
model = D3Model(mesh, out_dir = out_dir, use_periodic = True)

# the topography expressions :
surface = Expression(' - x[0] * tan(a)', a=a,
    element=model.Q.ufl_element())
bed = Expression(' - x[0] * tan(a) - 1000.0 + 500.0 * ' \
    + ' sin(2*pi*x[0]/L) * sin(2*pi*x[1]/L)',
    a=a, L=L, element=model.Q.ufl_element())

# mark the boundaries used for the integration :
model.calculate_boundaries()

# deform the mesh to the topography :
model.deform_mesh_to_geometry(surface, bed)

model.init_beta(1000) # traction
model.init_A(1e-16) # isothermal rate-factor

# just have to change the momentum physics to balance the other stress models :
mom = MomentumDukowiczBP(model)

```

Table 14.1: ISMIP-HOM stress-balance variables.

Variable	Value	Units	Description
$\dot{\epsilon}_0$	10^{-15}	a^{-1}	strain regularization
β	1000	$\text{kg m}^{-2}\text{a}^{-1}$	basal friction coef.
A	10^{-16}	$\text{Pa}^{-3}\text{a}^{-1}$	flow-rate factor
ℓ	8	km	width of domain
F_b	0	m a^{-1}	basal water discharge
a	0.5	$^\circ$	surface gradient mag.
\bar{B}	1000	m	average basal depth
b	500	m	basal height amp.
k_x	15	—	number of x divisions
k_y	15	—	number of y divisions
k_z	5	—	number of z divisions
N_e	6750	—	number of cells
N_n	1536	—	number of vertices

```

#mom = MomentumDukoviczStokesReduced(model)
#mom = MomentumDukoviczStokes(model)
mom.solve()

# solve the stress balance for the given Momentum instance :
F = StressBalance(model, momentum=mom)
F.solve()

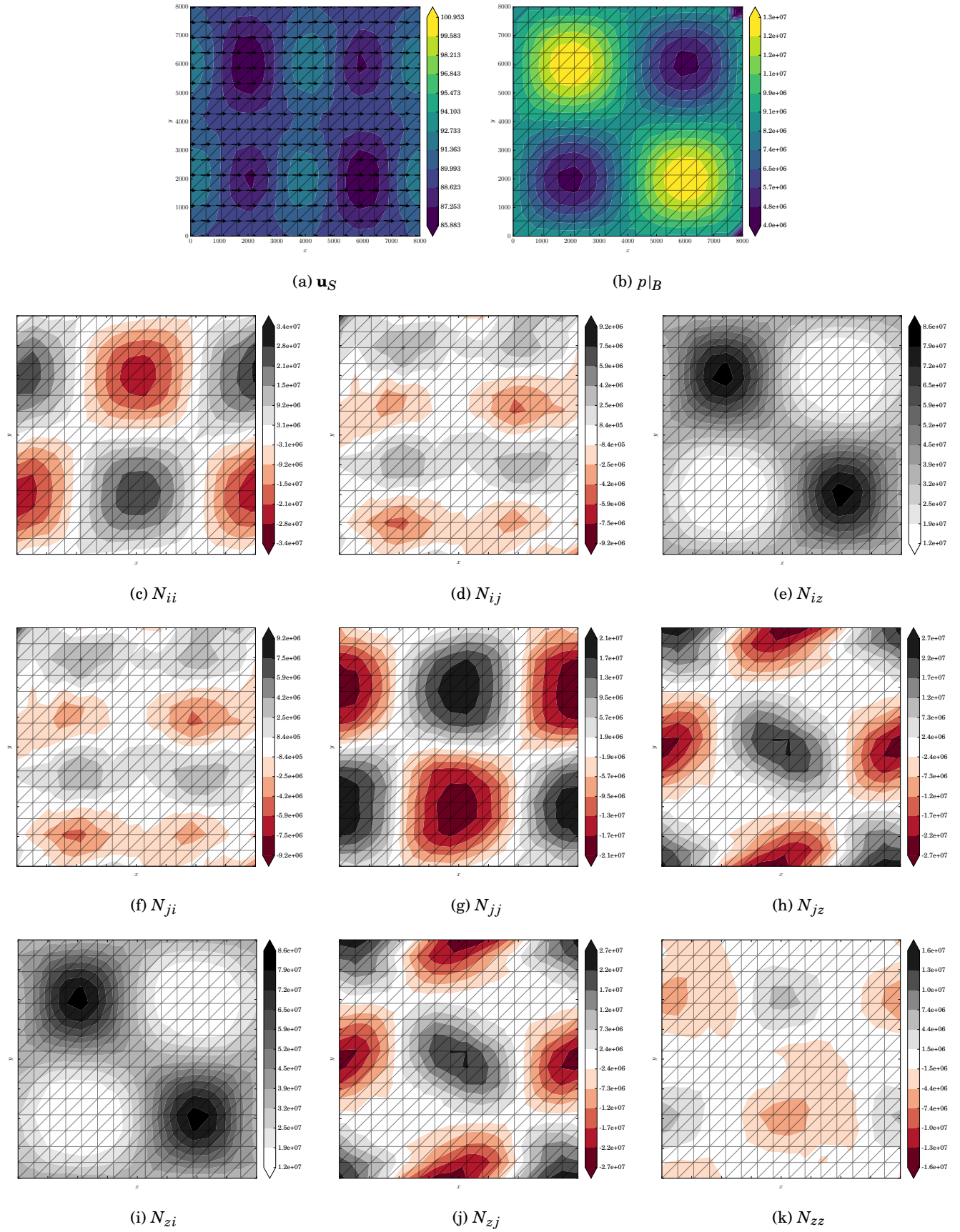
# save the resulting fields for plotting :
lst = [model.U3,
        model.p,
        model.N_ii,
        model.N_ij,
        model.N_iz,
        model.N_ji,
        model.N_jj,
        model.N_jz,
        model.N_zi,
        model.N_zj,
        model.N_zz,
        model.M_ii,
        model.M_ij,
        model.M_iz,
        model.M_ji,
        model.M_jj,
        model.M_jz,
        model.M_zi,
        model.M_zj,
        model.M_zz]

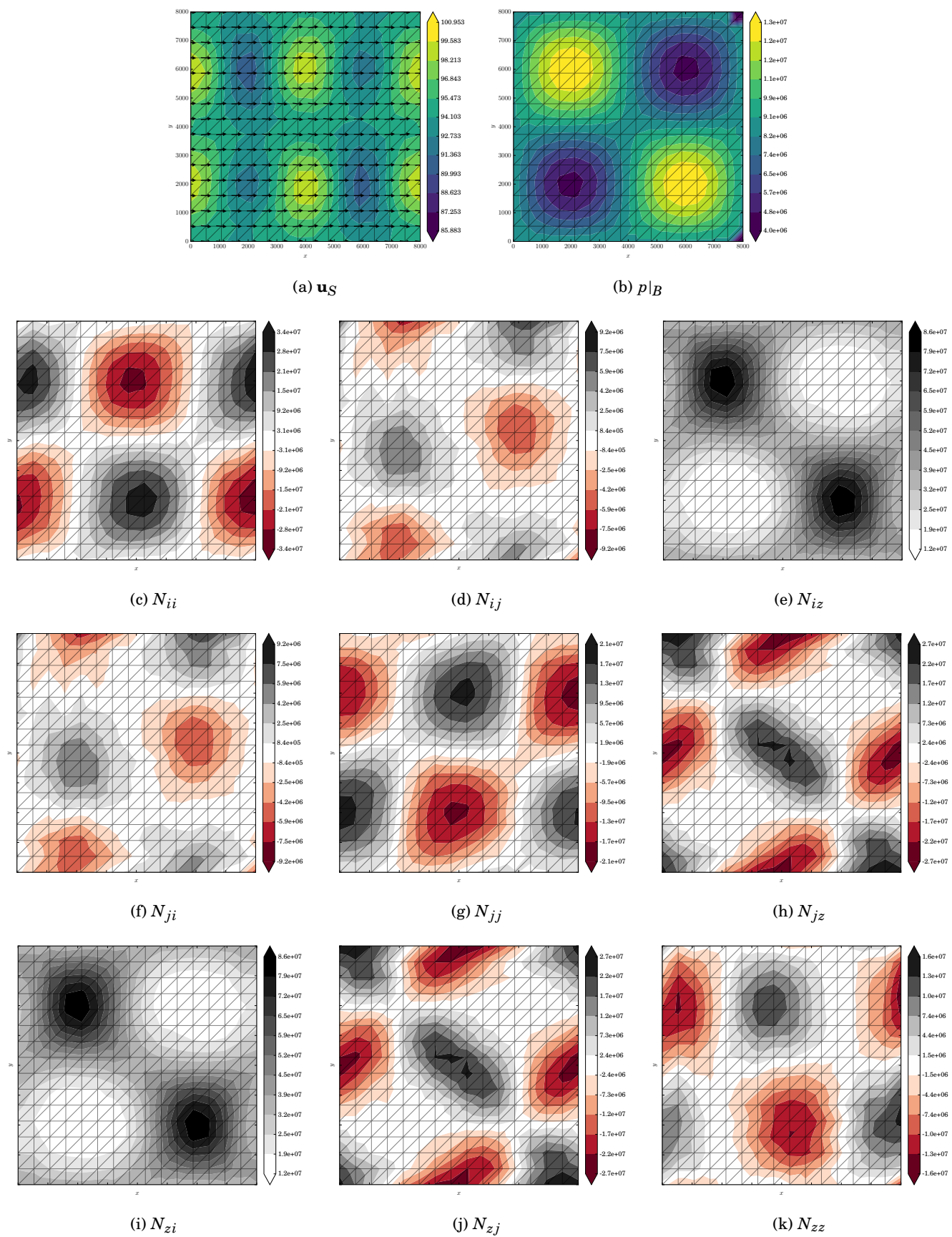
f = HDF5File(mpi_comm_world(), out_dir + 'FS.h5', 'w')

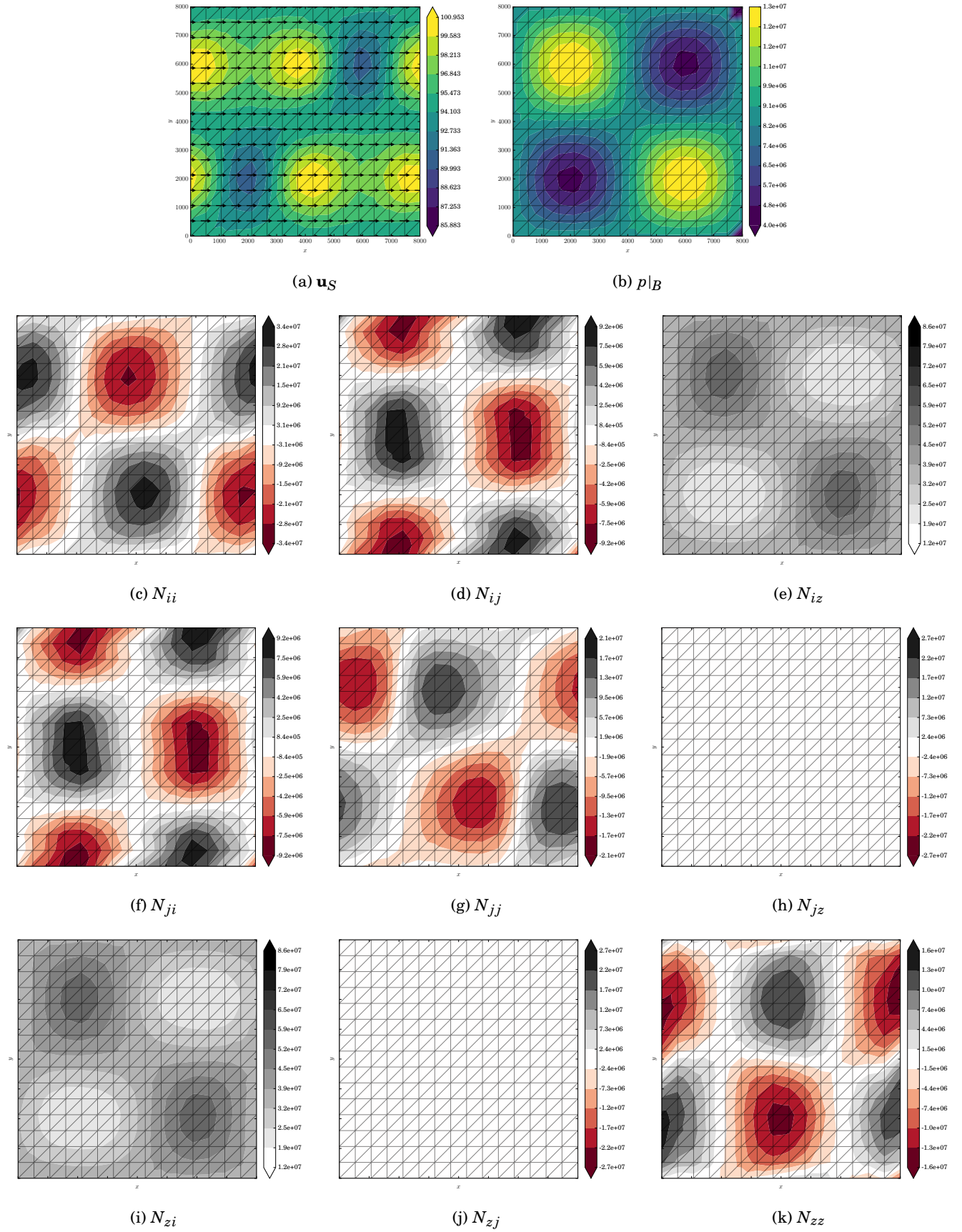
model.save_list_to_hdf5(lst, f)
model.save_subdomain_data(f)
model.save_mesh(f)

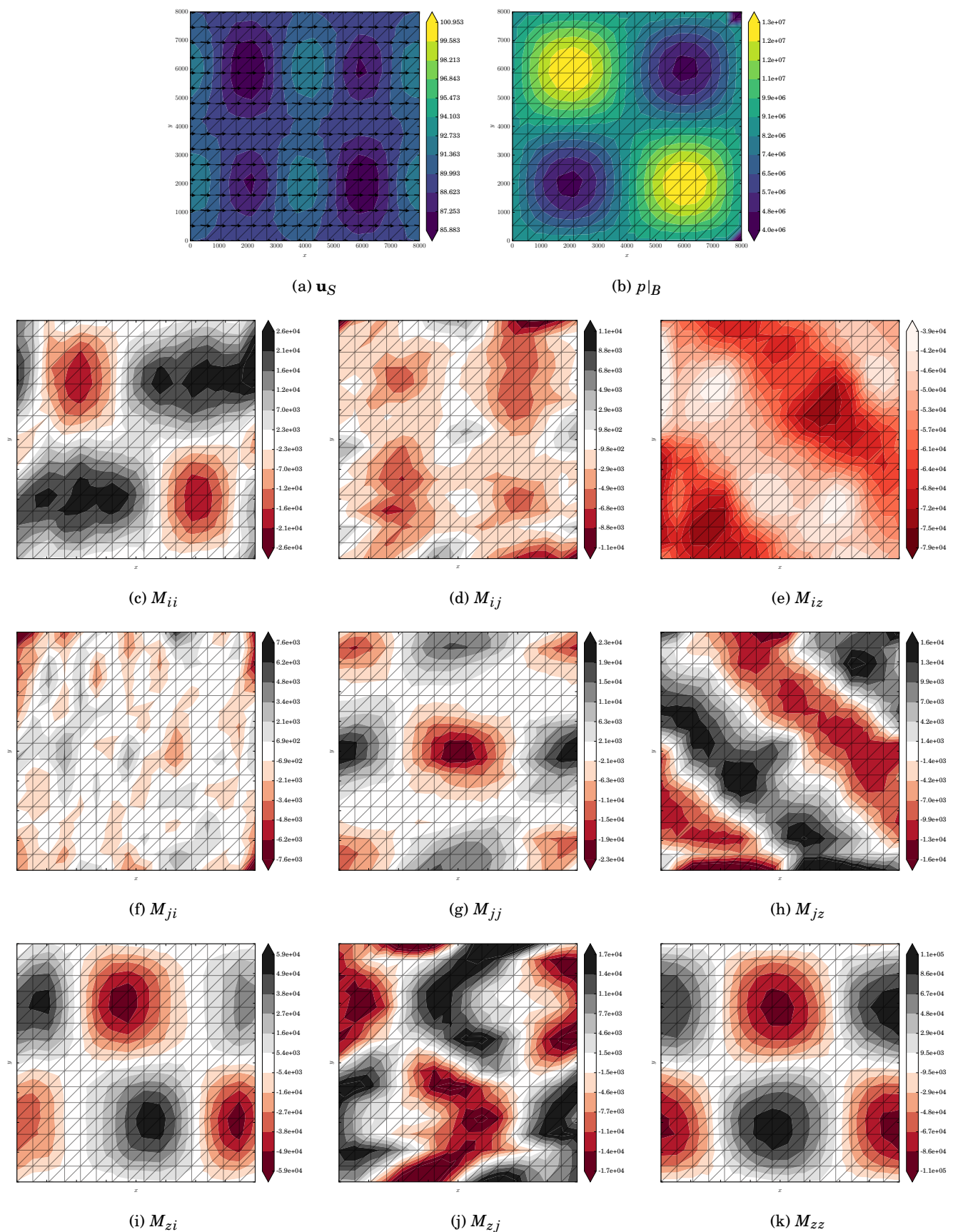
f.close()

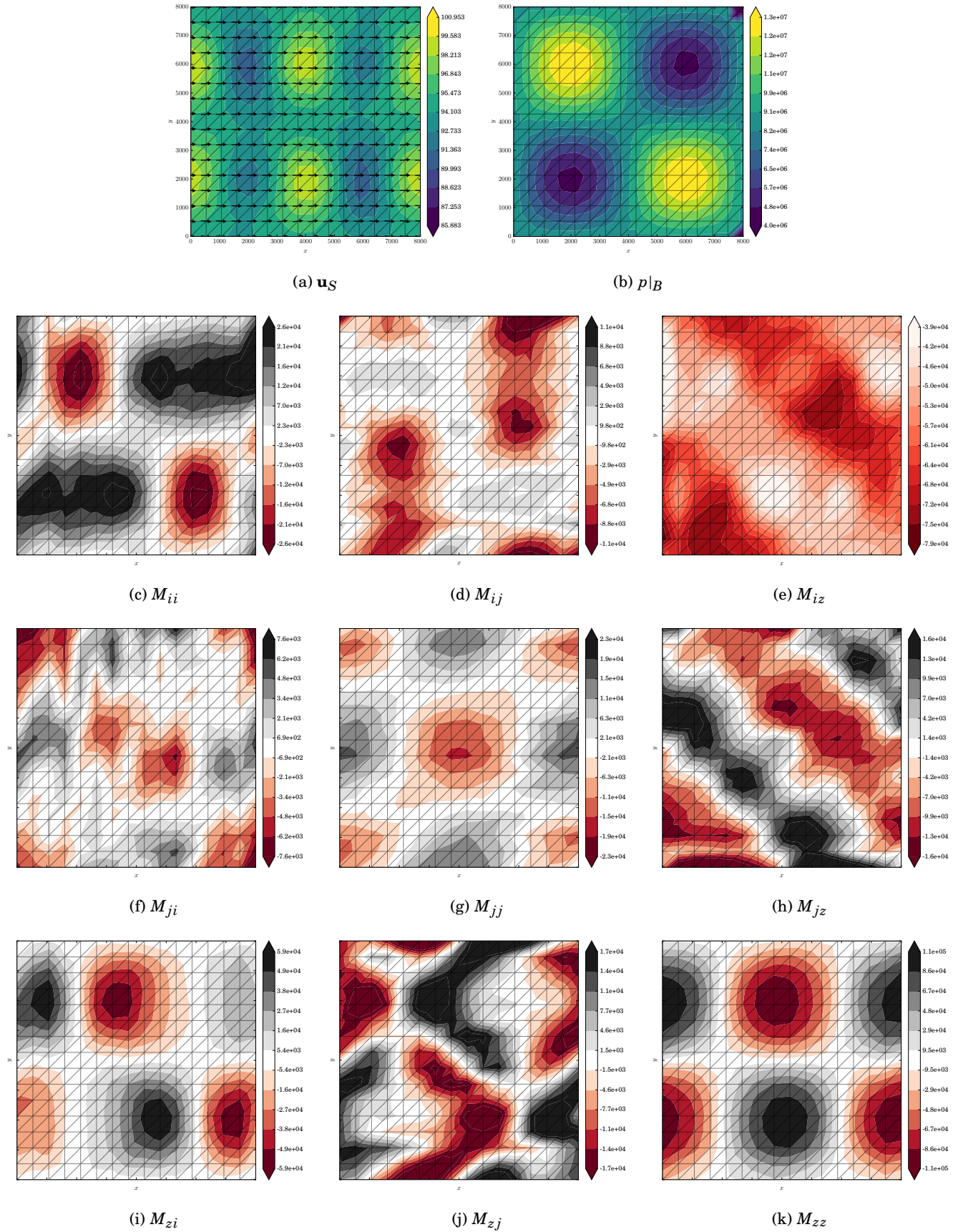
```

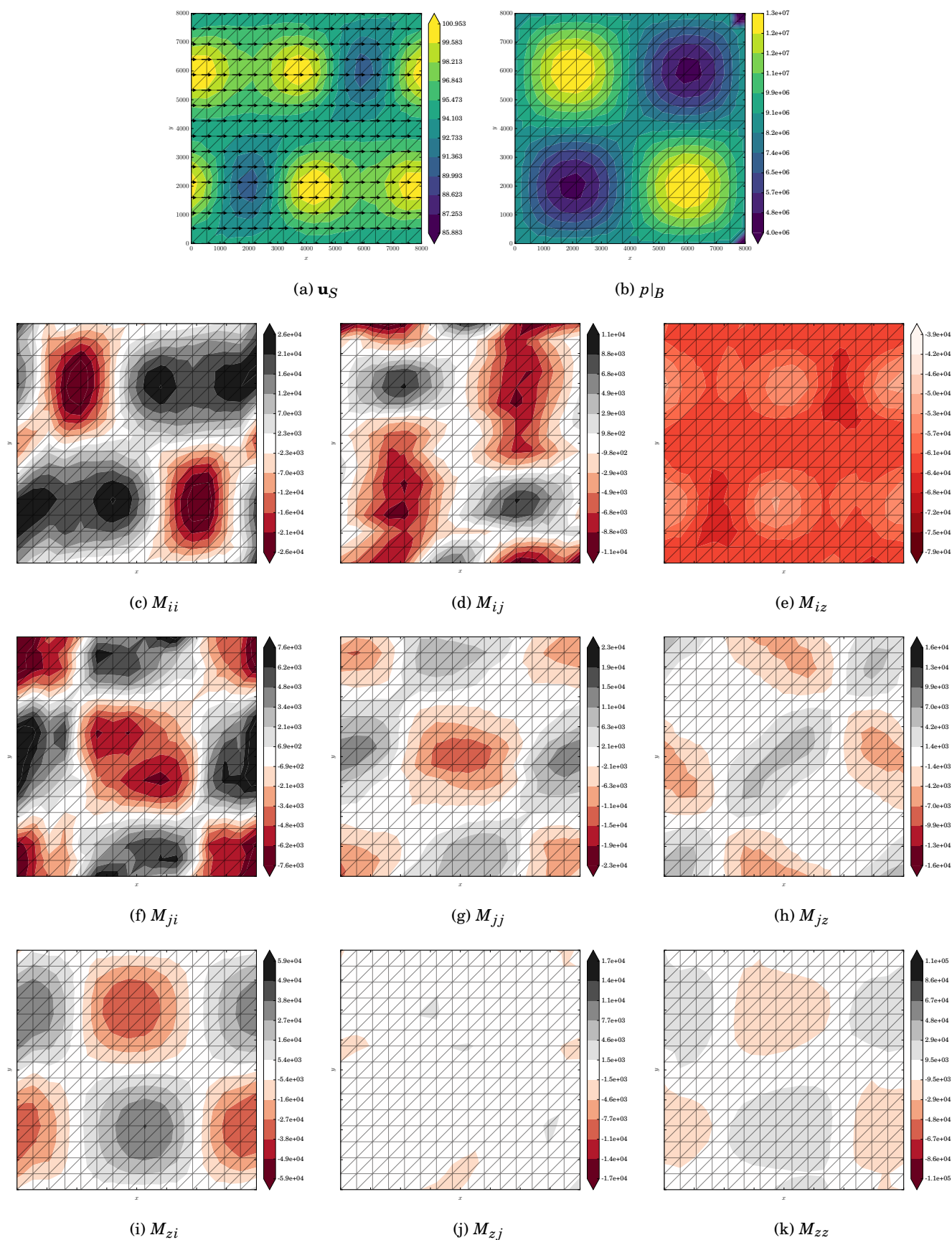
Figure 14.1: Full-Stokes membrane stress N_{kk} .

Figure 14.2: Reformulated-Stokes membrane stress N_{kk} .

Figure 14.3: First-order membrane stress N_{kk} .

Figure 14.4: Full-Stokes membrane stress balance M_{kk} .

Figure 14.5: Reformulated-Stokes membrane stress balance M_{kk} .

Figure 14.6: First-order membrane stress balance M_{kk} .

Chapter 15

Ice age

The total change in age with time is always equal to unity; for every step forward in time, the age of ice will age by an equivalent step. In order to quantify this change in the Eulerian coordinate system, the dependence of age on both time *and* space requires the evaluation of the material derivative of age a_{ge} with the *chain rule* :

$$\begin{aligned} \frac{da_{ge}}{dt} &= 1 \\ \frac{\partial a_{ge}}{\partial t} \frac{\partial t}{\partial t} + \frac{\partial a_{ge}}{\partial x} \frac{\partial x}{\partial t} + \frac{\partial a_{ge}}{\partial y} \frac{\partial y}{\partial t} + \frac{\partial a_{ge}}{\partial z} \frac{\partial z}{\partial t} &= 1 \\ \frac{\partial a_{ge}}{\partial t} + u \frac{\partial a_{ge}}{\partial x} + v \frac{\partial a_{ge}}{\partial y} + w \frac{\partial a_{ge}}{\partial z} &= 1 \\ \frac{\partial a_{ge}}{\partial t} + \mathbf{u} \cdot \nabla a_{ge} &= 1, \end{aligned} \quad (15.1)$$

where velocity $\mathbf{u} = [u \ v \ w]^T$ is the solution to one of the momentum-balance models described in Chapter 9.

In areas where the accumulation/ablation rate \dot{a} is positive, the ice-age on the surface will be new at the current time step. Hence a homogeneous, essential boundary condition is present there. Due to the fact that age equation (15.1) is hyperbolic, we cannot specify any other boundary conditions on the other ‘outflow’ surfaces (Hughes and Franca, 1987). Therefore, the only boundary condition is the essential condition

$$a_{ge} = 0 \quad \text{on } \Gamma_S|_{\dot{a} > 0} \quad \leftarrow \text{new snow.} \quad (15.2)$$

Note that in steady-state this equation becomes

$$\mathbf{u} \cdot \nabla a_{ge} = 1. \quad (15.3)$$

15.1 Variational form

This problem is purely advective, and as such the weak solution to this problem using standard Galerkin methods is numerically unstable (read to §5.5). To solve this issue, streamline upwind/Petrov-Galerkin (SUPG) stabilization (Brooks and Hughes, 1982) is applied, which has the effect of adding artificial diffusion to the variational form in areas of high velocity.

First, note that the intrinsic-time parameter for this problem is identical to (5.20) with $\xi = 1$ due to the fact that no

diffusion is present. Making the appropriate substitutions in (5.20) results in the intrinsic time parameter

$$\tau_{age} = \frac{h}{2\|\mathbf{u}\|}, \quad (15.4)$$

where h is the element size.

Therefore, using general stabilized form (5.16) with the operator $\mathcal{L}v = \mathbf{u} \cdot \nabla v$, SUPG operator (5.18), and intrinsic-time parameter (15.4), the stabilized variational form corresponding to steady-state age equation (15.3) consists of finding $a_{ge} \in S_E^h \subset \mathcal{H}_E^1(\Omega)$ (see trial space (1.10)) such that

$$\int_{\Omega} \mathbf{u} \cdot \nabla a_{ge} \phi \, d\Omega + \int_{\Omega} \tau_{age} \mathbf{u} \cdot \nabla \phi \, d\Omega = \int_{\Omega} \phi \, d\Omega \quad (15.5)$$

for all $\phi \in S_0^h \subset \mathcal{H}_{E_0}^1(\Omega)$ (see test space (1.11)), subject to essential boundary condition (15.3).

The implementation of this problem by CSLVR is shown in Code Listing 15.1.

Code Listing 15.1: CSLVR source code for the Age class.

```
from cslvr.physics import Physics

class Age(Physics):
    """
    Class for calculating the age of the ice in steady state.

    :Very simple PDE:
    .. math::
       \vec{u} \cdot \nabla A = 1

    This equation, however, is numerically challenging due to its being
    hyperbolic. This is addressed by using a streamline upwind Petrov
    Galerkin (SUPG) weighting.

    :param model : An instantiated 2D flowline ice :class:`src.model.Model`
    :param config : Dictionary object containing information on physical
                    attributes such as velocities, age, and surface climate
    """

    def __init__(self, model, solve_params=None, transient=False,
                 use_smb_for_ela=False, ela=None):
        """
        Set up the equations
        """
        s = ">::: INITIALIZING AGE PHYSICS :::"
        print_text(s, self.color())

        if type(model) != D3Model:
            s = ">>> Age REQUIRES A 'D3Model' INSTANCE, NOT %s <<<"
            print_text(s % type(model), 'red', 1)
            sys.exit(1)

        if solve_params == None:
            self.solve_params = self.default_solve_params()
        else:
            self.solve_params = solve_params

        # only need the cell size and velocity :
        h = model.h
        U = model.U3

        # Trial and test
        a = TrialFunction(model.Q)
        phi = TestFunction(model.Q)

        # Steady state
        if not transient:
            s = " - using steady-state -"
            print_text(s, self.color())

        # SUPG intrinsic time parameter :
        Unorm = sqrt(dot(U,U) + DOLFIN_EPS)
        tau = h / (2 * Unorm)
```

```

# the advective part of the operator :
def L(u): return dot(U, grad(u))

# streamlin-upwind/Petrov-Galerkin form :
self.a = + dot(U, grad(a)) * phi * dx \
+ inner(L(phi), tau*L(a)) * dx
self.L = + Constant(1.0) * phi * dx \
+ tau * L(phi) * dx

# FIXME: 3D model does not mesh-movement anymore.
else:
    s = " - using transient -"
    print_text(s, self.color())

# Time step
dt = model.time_step

# SUPG intrinsic-time (note subtraction of mesh velocity) :
U = as_vector([model.u, model.v, model.w - model.mhat])
Unorm = sqrt(dot(U,U) + DOLFIN_EPS)
tau = h / (2 * Unorm)

# midpoint value of age for Crank-Nicholson :
a_mid = 0.5*(a + self.ahat)

# SUPG intrinsic time parameter :
Unorm = sqrt(dot(U,U) + DOLFIN_EPS)
tau = h / (2 * Unorm)

# the advective part of the operator :
def L(u): return dot(U, grad(u))

# streamlin-upwind/Petrov-Galerkin form :
# FIXME: no a0 anymore
self.a = + (a - a0)/dt * phi * dx \
+ dot(U, grad(a_mid)) * phi * dx \
+ inner(L(phi), tau*L(a_mid)) * dx
self.L = + Constant(1.0) * phi * dx \
+ tau * L(phi) * dx

# form the boundary conditions :
if use_smb_for_ela:
    s = " - using adot (SMB) boundary condition -"
    print_text(s, self.color())
    self.bc_age = DirichletBC(model.Q, 0.0, model.ff_acc, 1)

else:
    s = " - using ELA boundary condition -"
    print_text(s, self.color())
    def above_ela(x,on_boundary):
        return x[2] > ela and on_boundary
    self.bc_age = DirichletBC(model.Q, 0.0, above_ela)

def solve(self):
    """
    Solve the system
    """
    model = self.model

    # Solve!
    s = " ::: solving age ::: "
    print_text(s, self.color())
    #solve(lhs(self.F) == rhs(self.F), model.age, self.bc_age)
    solve(self.a == self.L, self.age, self.bc_age)
    model.age.interpolate(self.age)
    print_min_max(model.age, 'age')

def solve_age(self, ahat=None, a0=None, uhat=None, what=None, vhat=None):
    """
    Solve the system

    :param ahat : Observable estimate of the age
    :param a0 : Initial age of the ice
    :param uhat : Horizontal velocity
    :param vhat : Horizontal velocity perpendicular to :attr:'uhat'
    :param what : Vertical velocity
    """
    # Assign values to midpoint quantities and mesh velocity
    if ahat:
        self.assign_variable(self.ahat, ahat)
        self.assign_variable(self.a0, a0)
        self.assign_variable(self.uhat, uhat)
        self.assign_variable(self.vhat, vhat)
        self.assign_variable(self.what, what)

    # Solve!
    s = " ::: solving age ::: "
    print_text(s, self.D3Model_color)
    solve(self.a == self.L, model.age, self.age_bc,
          annotate=False)
    #solve(self.a_a == self.a_L, self.age, self.age_bc, annotate=False)
    #self.age.interpolate(self.age)
    print_min_max(model.age, 'age')

```

Chapter 16

Application: Jakobshavn

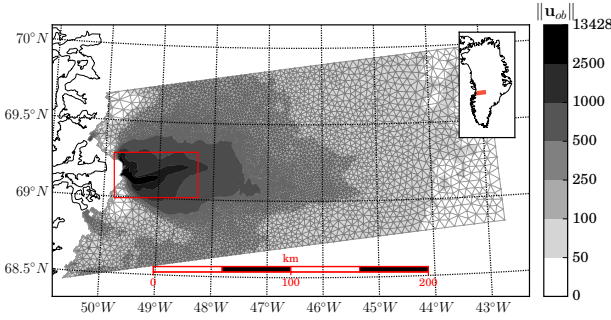


Figure 16.1: Surface velocity magnitude observations in m a^{-1} from Rignot and Mouginot (2012) over the Jakobshavn region highlighted in orange in the inlaid map.

The region surrounding Jakobshavn Glacier was chosen to test the applicability of the methods of Chapters 9, 10, and 12 to real-world data. To simplify computations while simultaneously producing high-resolution results, the domain was restricted to a rectangular region with Eastern-most boundary located close to the divide, with an along-flow width of approximately 120 km. A triangular mesh was first created with a minimum cell diameter of 500 m in the fastest-flowing regions and maximum cell diameter of 15 km in the slowest-flowing regions. This mesh was then extruded vertically into ten layers of tetrahedra and deformed to match the geometry supplied by Bamber et al. (2013) (see Code Listing 16.1 and Figures 16.1 and 16.2).

In order to retain the well-posedness of thermo-mechanical system (10.20), (10.38 – 10.43), (8.1 – 8.2), and (9.1 – 9.4), extra boundary conditions must be applied along the interior regions of ice having been cut by the specification of the mesh boundary, denoted Γ_D (Figure 8.1). These extra conditions are

$$\sigma \cdot \mathbf{n} = -f_c \mathbf{n} \quad \text{on } \Gamma_D \quad \leftarrow \text{full-Stokes cry'st'c str.} \quad (16.1)$$

$$\sigma_{BP} \cdot \mathbf{n} = \mathbf{0} \quad \text{on } \Gamma_D \quad \leftarrow \text{first-order cry'st'c str.} \quad (16.2)$$

$$\theta = \theta_S \quad \text{on } \Gamma_D \quad \leftarrow \text{Dirichlet,} \quad (16.3)$$

with cryostatic stress f_c defined by (9.31). As depicted by the $f_c \mathbf{n}$ vectors in Figure 8.1, full-Stokes stress condition (16.1) and first-order stress condition (16.2) represent the contribution of stress presented to the ice column by the surrounding

ice removed from the domain.

While it is unlikely that the energy is unchanging from the surface throughout the interior of the ice along the surface Γ_D as stated by ‘Dirichlet’ condition (16.3), the region of interest lies ≈ 75 km from this boundary, with very low velocity magnitude (Figure 16.1). Thus it is expected that the effect of this inconsistency is small.

To initialize the basal traction, we used the SIA approximate field β_{SIA} described in §12.2. In order to initialize flow-rate factor (10.22), the initial energy values θ^i throughout the interior Ω were initialized using quadratic energy (10.11) with initial water content $W^i(z) = 0$ and surface temperatures $T^i(x, y, z) = T_S(x, y), \forall z$ provided by Fausto et al. (2009). Also, pressure melting temperature (10.12) requires that pressure p be initialized; here we applied cryostatic pressure f_c in (9.31) such that $p^i(z) = f_c(z) = \rho g(S - z)$. Finally, the basal-water discharge F_b across the entire basal domain was initialized to zero.

The L^2 and logarithmic cost functional coefficients in (12.1), γ_1 and γ_2 , respectively, were determined as described in §12.5; by completing Algorithm 6 several times and adjusting their relative values such that at the end of the process their associated functionals were of approximately the same order (Figure 16.4).

An appropriate value for the regularization parameters γ_3 and γ_4 in momentum objective (12.1) was determined from an L-curve process (§12.3). First, it was noted that when performing this procedure for only one regularization term, i.e., setting one of either γ_3 or γ_4 to zero, this process resulted in choosing $\gamma_3 = \gamma_4 = 10$. We then took $\gamma_4 = 10$ and increased γ_3 until the Tikhonov regularization functional began to affect the regularization of optimal traction β^* , resulting in choosing $\gamma_3 = 0.1$.

Finally, the geothermal heat flux was set to the average Greenland value of $\bar{q}_{geo} = 4.2 \times 10^{-2} \text{ W m}^{-2}$ (Paterson, 1994) so as to minimize its effect on basal melt rate (10.36) and hence also basal water discharge F_b as expressed by basal energy flux (10.43). For a complete listing of initial variables and coefficients used, see Table 16.1. The CSLVR scripts used to generate the mesh, data, and results are shown in Code Listings 16.1, 16.2, and 16.3, respectively.

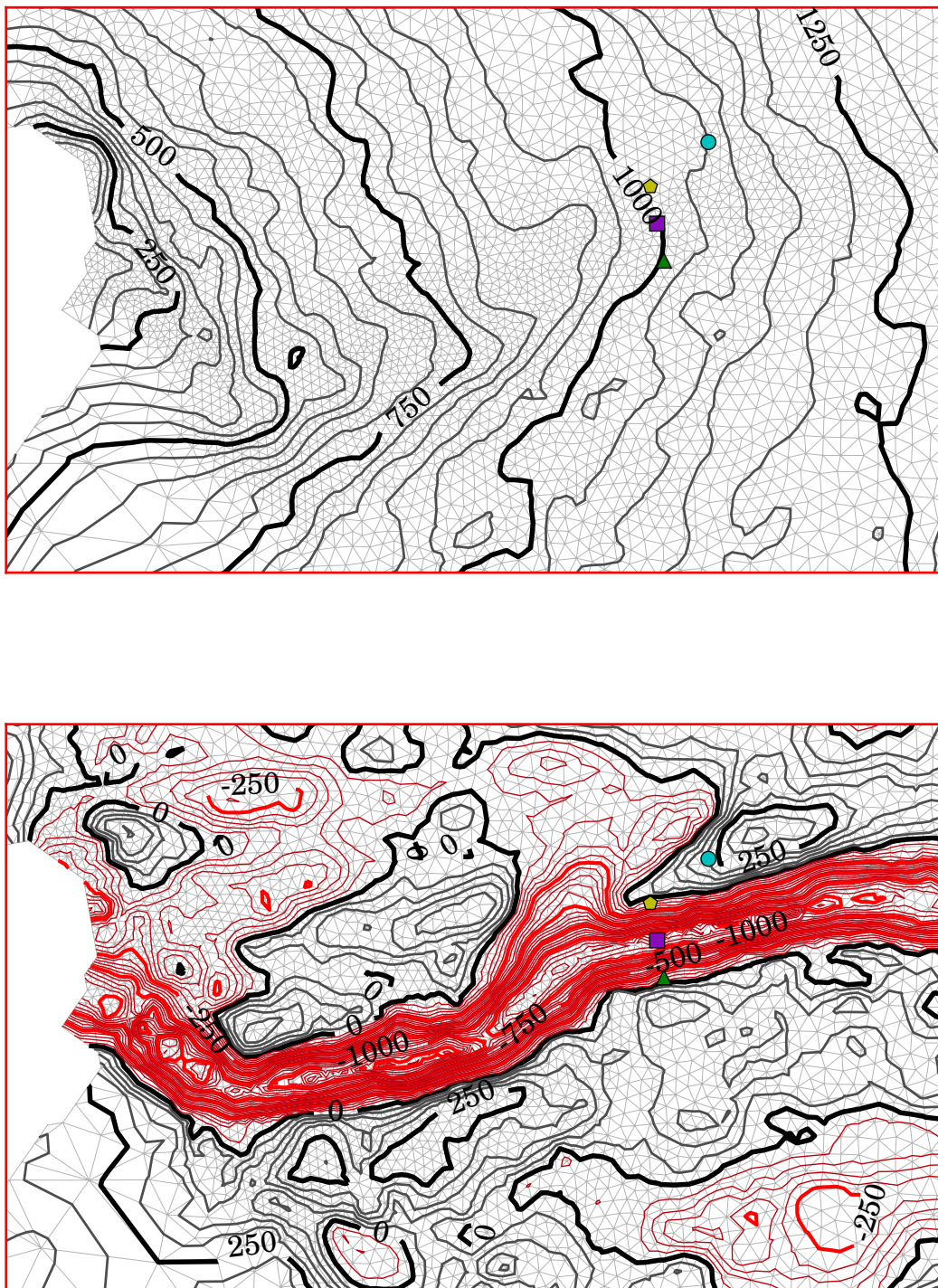


Figure 16.2: Topography of surface S in m (top) and basal topography in m (bottom) provided by Bamber et al. (2013) for the rectangular region outlined in red in Figure 16.1. Topography contours are spaced 50 m, with negative heights colored red. The colored points roughly correspond to the locations examined by Lüthi et al. (2002).

Table 16.1: Initial values used for Jakobshavn sim's.

Variable	Value	Units ¹	Description
D	0	m	ocean height
q_{geo}	4.2×10^{-2}	W m^{-2}	geothermal heat flux
ϵ_0	10^{-15}	a^{-1}	strain regularization
γ_1	5×10^3	$\text{kg m}^{-2} \text{a}^{-1}$	L^2 cost coefficient
γ_2	10^{-2}	J a^{-1}	log. cost coefficient
γ_3	10^{-1}	$\text{m}^6 \text{kg}^{-1} \text{a}^{-1}$	Tikhonov reg. coef.
γ_4	10	$\text{m}^6 \text{kg}^{-1} \text{a}^{-1}$	TV reg. coeff.
θ^i	$\theta(T_S, 0)$	J kg^{-1}	initial energy
β^i	β_{SIA}	$\text{kg m}^{-2} \text{a}^{-1}$	initial friction coef.
F_b^i	0	m a^{-1}	ini. basal water flux
p^i	f_c	Pa	initial pressure
k_0	10^3	–	energy flux coefficient

16.1 Results

The first set of results were generated using a maximum basal water content of $W_c = 0.01$, corresponding to the maximum observed in this area (Lüthi et al., 2002). To ensure convergence, Algorithm 6 was run for 10 iterations using 1000 iterations of momentum barrier problem (12.7) (Figure 16.4).

For each iteration of this simulation, the surface velocity mismatch between modeled results and observations supplied by Rignot and Mouginot (2012) remained at least one order of magnitude lower than the surface speed, with greatest error near the terminus (Figure 16.5). As evident by Figure 16.4, both regularization functionals (12.4, 12.5) and the L^2 cost functional (12.2) of momentum objective (12.1) reach approximately the same respective minimums at the end of each iteration, while the logarithmic cost functional (12.3) – which is most affected by velocity mismatches in slower regions of flow distant from the terminus – continues to decrease. This may be due to the fact that for each iteration, both basal-melting rate M_b and basal traction β remain quite low near the region of fast flow nearest the terminus, and thus L^2 cost (12.2) is little affected by changes in internal energy.

It was observed that for each iterative change in basal traction β – and thus also basal melt-rate (10.36) and internal friction (8.11) – the optimization procedure for basal-water discharge F_b would reach a value required to flush out water generated by both of these sources. As a result of this, at each iteration the basal water content remained close to W_c and the vertically averaged water content \bar{W} remained in all but a few areas below 5% (Figure 16.5).

The column percentage of temperate ice resulting from this simulation is on average about twice that reported by Lüthi et al. (2002), despite the imposition of an approximately 10° K lower surface temperature. This is likely due to both the improper specification of the CTS constraints, as explained by Blatter and Greve (2015), and the low vertical resolution of the mesh (Figure 16.3). The sharp decline in temperature near the surface, followed by the sharp increase at the next vertex of the mesh, suggests that the thermal gradient near the surface is creating numerical oscillations. In such a case,

increasing the vertical resolution of the mesh will reduce the instability and result in a higher-magnitude temperature gradient near the surface, hence reducing the temperature to a level closer to expectations.

Remarkably, nearly the entire column of ice is temperate along the shear margins nearest the terminus (Figure 16.5). As a result, basal traction β must increase in order to compensate for the approximately three-fold enhancement of flow-rate factor (10.22) and relatively low surface speed there. Additionally, the basal water discharge optimization process does not appear to compensate for the relatively high basal water content of these regions; it may be necessary to append energy objective (10.72) with an additional term favoring areas with largest misfit, similar to momentum objective (12.1). Finally, we note that an entirely different ice-flow model applies to the shear margins due to the extreme levels of strain-heat and damaged ice in this region.

For an additional test, the data-assimilation process described by Algorithm 6 was performed once more for ten iterations with two simulations; one using a maximum basal water content $W_c = 0.03$ and one using $W_c = 0.01$, in the hope of comparison. The ratio of basal traction fields derived using maximum water content $W_c = 0.01$ and $W_c = 0.03$ is shown in Figure 16.6. While the traction appears to be up to six times greater along the main trench, it is important to note that the friction is quite small in this region (Figure 16.5) and thus has little effect on the velocity field. However, the traction along the flanks of the trench approximately 20 km inland are up to half as large when the maximum basal water content is reduced from 3% to 1%. This may be explained by both the reduction in height of the CTS and the reduction in water content to a point below that required by the empirically-constrained water content relation W_f in rate factor (10.22).

16.2 Conclusion

The methods presented in §10.6 and §12.2 offer a new approach to approximate the energy and momentum distributions of an ice-sheet or glacier. This method, having been derived from established balance equations, provides the ability to match energy distributions to observations of internal-water content. This method results in a consistent estimate of velocity, energy, basal-water discharge, and basal traction. Applying this procedure over the region of Jakobshavn indicated that altering the maximum-allowed basal-water content has a significant effect on basal traction values derived by data-assimilation methods.

Code Listing 16.1: CSLVR script used to generate the mesh for the Jakobshavn simulation.

```
from csivr import *
from pylab import *
from scipy.interpolate import RectBivariateSpline

# =====
# data preparation :
out_dir = 'dump/meshes/'
mesh_name = 'jakobshavn_3D_small_block'

# get the data :
bamber = DataFactory.get_bamber()
rignot = DataFactory.get_rignot()

# process the data into the format used by CSLVR :
```

```
dbm      = DataInput(bamber,   gen_space=False)
drg      = DataInput(rignot,   gen_space=False)

# the 'Rignot' dataset is defined with a different projection :
drg.change_projection(dbm)

# get surface velocity magnitude :
U_ob = sqrt(drg.data['vx']**2 + drg.data['vy']**2 + 1e-16)
drg.data['U_ob'] = U_ob

# form field from which to refine :
#-----

# this method will create a new data field with key 'ref' that we will
# use to define the cell diameter, with maximum diameter umax, minimum
# diameter umin, and varied inversely proportional to the first argument;
# we want the cell size to be small where the ice is moving fast, and large
# where the ice is moving slowly :
drg.rescale_field('U_ob', 'ref', umin=500.0, umax=300000.0, inverse=True)

# eliminate just the edge of the mask so that we can properly interpolate
# the geometry to the terminus :
l = dbm.data['lat_mask']
dbm.data['mask'][L > 0.0] = 0

# generate the contour :
#-----

# the meshgenerator will create a mesh from the 'Bamber' data :
m = MeshGenerator(dbm, mesh_name, out_dir)

# generate a contour around the land-ice mask 'mask', skipping zero points :
m.create_contour('mask', zero_cntr=0.0001, skip_pts=0)

# coordinates of the box in x,y projection coordinates that we'll
# slice out of the continent :
x1 = -500000; y1 = -2190000
x2 = -150000; y2 = -2320000

# the x,y contour of the box 'slice' :
new_cont = array([[x1, y1],
                  [x2, y1],
                  [x2, y2],
                  [x1, y2],
                  [x1, y1]])

# get the intersection of the contour of the entire continent and the box :
m.intersection(new_cont)

# make sure that no paths intersect of the new contour :
m.eliminate_intersections(dist=20)

# transform the contour to the projection used by the 'Rignot' dataset :
m.transform_contour(rignot)

# make sure no two points lie directly on top of each other :
m.check_dist()

# save the resulting, perfect contour :
m.write_gmsh_contour(boundary_extend=False)

# you can plot the contour to make sure its perfect too :
#m.plot_contour()

# we have a 3D mesh here, so we extrude it any uniform distance vertically,
# with a number of layers :
m.extrude(h=100000, n_layers=10)

# close that .geo contour file so we don't corrupt any data :
m.close_file()

# mesh refinement :
#-----

# the MeshRefiner class uses a data array to set the cellsize to something
# we want, here the observed velocity speed of the 'Rignot' dataset :
ref_bm = MeshRefiner(drg, 'ref', gmsh_file_name = out_dir + mesh_name)

# we need the background field from which we refine :
a, aid = ref_bm.add_static_attractor()
ref_bm.set_background_field(aid)

# this refines off of the background field we just set :
ref_bm.finish(gui = False, out_file_name = out_dir + mesh_name)

# and convert to the xml.gz file utilized by FEniCS :
ref_bm.convert_msh_to_xml()
```

Code Listing 16.2: CSLVR script used to generate the data used by Code Listing 16.3.

```
from csivr    import *
from fenics   import *
from pylab    import *

out_dir      = 'dump/vars_jakobshavn_small/'

# collect the raw data :
searise      = DataFactory.get_searise()
bamber       = DataFactory.get_bamber(1.0)
rignot       = DataFactory.get_rignot()

# define the mesh :
mesh = Mesh('dump/meshes/jakobshavn_3D_small_block.xml.gz')

# create data objects to use with varglas :
dsr      = DataInput(searise,   mesh=mesh)
dbm      = DataInput(bamber,    mesh=mesh)
drg      = DataInput(rignot,    mesh=mesh)

# change the projection of all data to be the same as the mesh :
dsr.change_projection(drg)
dbm.change_projection(drg)

# get the expressions used by varglas :
S      = dbm.get_expression('S',      near=False) # surface height
B      = dbm.get_expression('B',      near=False) # bed height
M      = dbm.get_expression('mask',   near=True)  # shelf mask
L      = dbm.get_expression('lat_mask', near=True) # terminus mask
adot   = dsr.get_expression('adot',   near=False) # acc/abl function
q_geo  = dsr.get_expression('q_geo',   near=False) # geothermal heat
T_s    = dsr.get_expression('T',      near=False) # surface temperature
v_ob   = drg.get_expression('vx',     near=False) # x-comp. surface velocity
v_ob   = drg.get_expression('vy',     near=False) # y-comp. surface velocity
U_msk  = drg.get_expression('mask',   near=True)  # where U obs. are pres.

# create the 3D model :
```

```
model = D3Model(mesh=mesh, out_dir=out_dir)

# deform the mesh to match the geometry provided by the 'Bamber' dataset :
model.deform_mesh_to_geometry(S, B)

# calculate the boundaries appropriately. The lat_mask marks the exterior
# cliff edges of the mesh, U_mask marks areas without U observations,
# adot creates the age equations surface boundary condition, and
# mark_divide is set to True to inform the model to mark the
# boundaries interior to the domain.
model.calculate_boundaries(mask=M, lat_mask=L, U_mask=U_msk, adot=adot,
                           mark_divide=True)

# initialize the observations :
model.init_T_surface(T_s)
model.init_q_geo(q_geo)
model.init_U_ob(v_ob, v_ob)

# these area all data fields we'll need to perform the data-assimilation
# procedure :
lst = [model.S,
       model.B,
       model.mask,
       model.q_geo,
       model.T_surface,
       model.adot,
       model.u_ob,
       model.v_ob,
       model.U_mask,
       model.lat_mask]

# the container for the data :
f = HDF5File(mpi_comm_world(), out_dir + 'state.h5', 'w')

# save all the data :
model.save_list_to_hdf5(lst, f)

# save the facet and cell markers generated by calculate_boundaries() :
model.save_subdomain_data(f)

# save the geometry-deformed mesh :
model.save_mesh(f)

f.close()
```

Code Listing 16.3: CSLVR script used to simultaneously assimilate data u_{ob} and W_c .

```
from csivr    import *
from scipy    import random
from fenics   import *
from dolfin_adjoint import *
import sys

# set the relevant directories :
var_dir = 'dump/vars_jakobshavn_small/' # directory from gen_vars.py
out_dir = 'dump/jakob_small/inversion_Wc_0.03/'

# create HDF5 files for saving and loading data :
fmeshes = HDF5File(mpi_comm_world(), var_dir + 'submeshes.h5', 'r')
fdata    = HDF5File(mpi_comm_world(), var_dir + 'state.h5', 'r')

# create 3D model for stokes solves :
d3model = D3Model(fdata, out_dir)

# init subdomains and boundary meshes :
d3model.set_subdomains(fdata)
d3model.set_srf_mesh(fmeshes)
d3model.set_bed_mesh(fmeshes)
d3model.set_dvd_mesh(fmeshes)

# initialize the 3D model vars :
d3model.init_S(fdata) # surface topography
d3model.init_B(fdata) # basal topography
d3model.init_mask(fdata) # ice-shelf mask (here no shelf)
d3model.init_q_geo(d3model.ghf) # geothermal heat flux
d3model.init_T_surface(fdata) # surface temperature
d3model.init_adot(fdata) # accumulation/ablation
d3model.init_U_ob(fdata, fdata) # observed velocity
d3model.init_U_mask(fdata) # where U_ob data is missing
d3model.init_W(0.0) # no water content initially
d3model.init_Wc(0.03) # max allowed basal water content
d3model.init_T(d3model.T_surface) # initial temperature
d3model.init_k_0(1e-3) # non-advective enthalpy flux
d3model.solve_hydrostatic_pressure() # init to hydrostatic pressure
d3model.form_energy_dependent_rate_factor() # we are thermo-mechanical

# this commented-out section is how we can initialize the data to a previous
# state :

#frstrt = HDF5File(mpi_comm_world(), out_dir + '01/inverted.h5', 'r')
#d3model.init_T(frstrt)
#d3model.init_B(frstrt)
#d3model.init_W(frstrt)
#d3model.init_Fb(frstrt)
#d3model.init_alpha(frstrt)
#d3model.init_U(frstrt)
#d3model.init_p(frstrt)
#d3model.init_theta(frstrt)

# gaps in surface velocity data will create problems for our SIA-approximated
# initial fraction values (beta {SIA}). To correct this, we use instead of
# the surface velocity, the balance velocity, in these regions. See chapter
# on data assimilation for more info on this field
#-----

# create a 2D model for balance-velocity :
bedmodel = D2Model(d3model.bedmesh, out_dir)

# initialize the field we need from the 3D mesh :
bedmodel.assign_submesh_variable(bedmodel.S, d3model.S)
bedmodel.assign_submesh_variable(bedmodel.B, d3model.B)
bedmodel.assign_submesh_variable(bedmodel.adot, d3model.adot)

# solve the balance velocity (see the appropriate Chapter) :
bv = BalanceVelocity(bedmodel, kappa=5.0, stabilization_method = 'GLS')
bv.solve_direction_of_flow([bedmodel.S.dx(0), bedmodel.S.dx(1)])
bv.solve(annotate=False)

# assign the balance velocity to the 3D model's bed :
d3model.assign_submesh_variable(d3model.d.x, bedmodel.d.x)
d3model.assign_submesh_variable(d3model.d.y, bedmodel.d.y)
d3model.assign_submesh_variable(d3model.Ubar, bedmodel.Ubar)

# extrude the bed values up the ice column :
d_x_e = d3model.vert_extrude(d3model.d.x, d='up')
d_y_e = d3model.vert_extrude(d3model.d.y, d='up')
Ubar_e = d3model.vert_extrude(d3model.Ubar, d='up')

# set the appropriate variable to be the function extruded :
```

```

d3model.init_d_x(d_x_e)
d3model.init_d_y(d_y_e)
d3model.init_Ubar(Ubar_e)

# generate initial traction field from the SIA approximation :
d3model.init_beta_SIA()

# the assimilation process uses the first-order model :
mom = MomentumDukowiczBP(d3model, linear=False)

# the TMC process uses full-Stokes :
momTMC = MomentumDukowiczStokes(d3model, linear=False)

# the energy-balance physics :
nrg = Enthalpy(d3model, momTMC, transient=False, use_lat_bc=True)

# this commented out section allows you to restart a simulation from a previous
# momentum optimization :

#frstrt = HDF5File(mpi_comm_world(), out_dir + '02/u_opt.h5', 'r')
#d3model.set_out_dir(out_dir + '02/')
#d3model.init_U(frstrt)
#d3model.init_beta(frstrt)

# thermo-solve callback function is called after every TMC iteration :
def tmc_cb_ftn():
    nrg.calc_PE() #avg=True
    nrg.calc_vert_avg_strain_heat()
    nrg.calc_vert_avg_W()
    nrg.calc_temp_rat()

# post-adjoint-iteration callback function is called after the TMC data
# assimilation process is finished :
def adj_post_cb_ftn():
    # solve for optimal vertical velocity :
    mom.solve_vert_velocity(annotate=False)

# after every completed adjoining, save the state of these functions :
adj_save_vars = [d3model.T,
                 d3model.W,
                 d3model.Fb,
                 d3model.Nb,
                 d3model.alpha,
                 d3model.alpha_int,
                 d3model.PE,
                 d3model.Wbar,
                 d3model.Qbar,
                 d3model.temp_rat,
                 d3model.U3,
                 d3model.p,
                 d3model.beta,
                 d3model.theta]

# save these variables after the momentum optimization :
u_opt_save_vars = [d3model.beta, d3model.U3]

# save these variables after the energy optimization :
w_opt_save_vars = [d3model.Fb, d3model.theta]

# form the momentum cost functional :
mom.form_obj_ftn(integral=d3model.GAMMA_U_GND, kind='log_L2_hybrid',
                g1=0.01, g2=5000)

# form the traction-regularization functional :
mom.form_reg_ftn(d3model.beta, integral=d3model.GAMMA_B_GND,
                kind='TV_Tik_hybrid', alpha_tik=1e-1, alpha_tv=10.0)
#mom.form_reg_ftn(d3model.beta, integral=d3model.GAMMA_B_GND,
#                kind='TV', alpha=10.0)
#mom.form_reg_ftn(d3model.beta, integral=d3model.GAMMA_B_GND,
#                kind='Tikhonov', alpha=1e-6)

# form the objective functional for water-flux optimization :
nrg.form_cost_ftn(kind='L2')

# keyword arguments to the energy-optimization function :
wop_kwargs = {'max_iter' : 350,
              'bounds' : (0.0, 100.0),
              'method' : 'ipopt',
              'adj_save_vars' : w_opt_save_vars,
              'adj_callback' : None}

# keyword arguments to the TMC function :
tmc_kwargs = {'momentum' : momTMC,
              'energy' : nrg,
              'wop_kwargs' : wop_kwargs,
              'callback' : tmc_cb_ftn,
              'atol' : 1e2,
              'rtol' : 1e0,
              'max_iter' : 5,
              'iter_save_vars' : None,
              'post_tmc_save_vars' : None,
              'starting_i' : 1}

# keyword arguments to the momentum optimization function :
uop_kwargs = {'control' : d3model.beta,
              'bounds' : (1e-5, 1e7),
              'method' : 'ipopt',
              'max_iter' : 1000,
              'adj_save_vars' : u_opt_save_vars,
              'adj_callback' : None,
              'post_adj_callback' : adj_post_cb_ftn}

# keyword arguments for the TMC data assimilation function :
ass_kwargs = {'momentum' : mom,
              'beta_i' : d3model.beta.copy(True),
              'max_iter' : 10,
              'tmc_kwargs' : tmc_kwargs,
              'uop_kwargs' : uop_kwargs,
              'atol' : 1.0,
              'rtol' : 1e-4,
              'initialize' : True,
              'incomplete' : True,
              'post_iter_save_vars' : adj_save_vars,
              'post_ini_callback' : None,
              'starting_i' : 1}

# assimilate ! :
d3model.assimilate_U_ob(**ass_kwargs)

# or simply optimize the traction coefficient without thermo :
#mom.optimize_U_ob(**uop_kwargs)

# or only thermo_solve :
#d3model.thermo_solve(**tmc_kwargs)

```

Code Listing 16.4: CSLVR script used calculate the stress-balance for the Jakobshavn simulation results as depicted in Figures 16.7 and 16.8.

```

from csylvr import *

# set the relevant directories :
base_dir = 'dump/jakob_small/inversion_Wc_0.01/10/'
in_dir = base_dir
out_dir = base_dir + 'stress_balance/'
var_dir = 'dump/vars_jakobshavn_small/'

# create HDF5 files for saving and loading data :
fdata = HDF5File(mpi_comm_world(), var_dir + 'state.h5', 'r')
fin = HDF5File(mpi_comm_world(), in_dir + 'inverted_10.h5', 'r')
fout = HDF5File(mpi_comm_world(), in_dir + 'stress.h5', 'w')

# create 3D model for stokes solves :
model = D3Model(fdata, out_dir)

# init subdomains :
model.set_subdomains(fdata)

# initialize the 3D model vars :
model.init_S(fdata)
model.init_B(fdata)
model.init_mask(fdata)
model.init_adot(fdata)
model.init_beta(fin)
model.init_U(fin)
model.init_p(fin)
model.init_T(fin)
model.init_W(fin)
model.init_theta(fin)
model.form_energy_dependent_rate_factor()
model.calc_A()

mom = MomentumDukowiczStokes(model)
F = StressBalance(model, momentum=mom)
F.solve()

model.save_hdf5(model.N_ii, f=fout)
model.save_hdf5(model.N_ij, f=fout)
model.save_hdf5(model.N_iz, f=fout)
model.save_hdf5(model.N_ji, f=fout)
model.save_hdf5(model.N_jj, f=fout)
model.save_hdf5(model.N_jz, f=fout)
model.save_hdf5(model.N_zi, f=fout)
model.save_hdf5(model.N_zj, f=fout)
model.save_hdf5(model.N_zz, f=fout)

model.save_hdf5(model.M_ii, f=fout)
model.save_hdf5(model.M_ij, f=fout)
model.save_hdf5(model.M_ji, f=fout)
model.save_hdf5(model.M_jj, f=fout)
model.save_hdf5(model.M_jz, f=fout)
model.save_hdf5(model.M_zi, f=fout)
model.save_hdf5(model.M_zj, f=fout)
model.save_hdf5(model.M_zz, f=fout)

fout.close()

```

Code Listing 16.5: CSLVR script used generate Figures 16.1 and 16.2.

```

from csylvr import *
from fenics import *
import matplotlib.pyplot as plt
import numpy as np
import sys

# set the relevant directories :
var_dir = 'dump/vars_jakobshavn_small/'
out_dir = '../images/internal_energy/jakob_results/inversion_Wc_0.01/'

if not os.path.exists(out_dir):
    os.makedirs(out_dir)

# not deformed mesh :
mesh = Mesh('dump/meshes/jakobshavn_3D_small_block.xml.gz')

# create 3D model for stokes solves :
d3model = D3Model(mesh, out_dir)

# retrieve the bed mesh :
d3model.form_srf_mesh()

# create 2D model for balance velocity :
srfmodel = D2Model(d3model.srfmesh, out_dir)

# open the hdf5 file :
fdata = HDF5File(mpi_comm_world(), var_dir + 'state.h5', 'r')

# initialize the variables :
d3model.init_S(fdata)
d3model.init_B(fdata)
d3model.init_U_ob(fdata, fdata)

# 2D model gets balance-velocity appropriate variables initialized :
srfmodel.assign_submesh_variable(srfmodel.S, d3model.S)
srfmodel.assign_submesh_variable(srfmodel.B, d3model.B)
srfmodel.assign_submesh_variable(srfmodel.U_ob, d3model.U_ob)

# =====
bamber = DataFactory.get_bamber()
rignot = DataFactory.get_rignot()

bamber['Bo'][bamber['Bo'] == -9999] = 0.0
bamber['S'][bamber['mask'] == 0] = 0.0
bamber['S'][bamber['S'] < 500] = 500.0

dbm = DataInput(bamber, gen_space=False)
drg = DataInput(rignot, gen_space=False)

dbm.change_projection(drg)

bc = '#880cbc'

lat_1 = 69.210
lat_2 = 69.168
lon_1 = -48.78
lon_2 = -48.759

dlat = (lat_2 - lat_1) / 2.0
dlon = (lon_2 - lon_1) / 2.0

lat_3 = lat_1 + dlat
lon_3 = lon_1 + dlon

```

```

lat_a = [ 69.235, lat_1, lat_2, lat_3]
lon_a = [-48.686944, lon_1, lon_2, lon_3]
color_a = ['c', 'y', 'g', 'bc']
style_a = ['o', 'p', '-', 's']

plot_pts = {'lat' : lat_a,
            'lon' : lon_a,
            'style' : style_a,
            'color' : color_a}

zoom_box_kwargs = {'zoom' : 4, # amount to zoom
                  'loc' : 9, # location of box
                  'loc1' : 3, # loc of first line
                  'loc2' : 4, # loc of second line
                  'llcrnrlon' : -51, # first x-coord
                  'llcrnrlat' : 68.32, # first y-coord
                  'urcrnrlon' : 42.5, # second x-coord
                  'urcrnrlat' : 70.1, # second y-coord
                  'plot_zoom_scale' : False, # draw the scale
                  'scale_font_color' : bc, # scale font color
                  'scale_length' : 20, # scale length in km
                  'scale_loc' : 1, # 1=top, 2=bottom
                  'plot_grid' : True, # plot the triangles
                  'axes_color' : bc, # color of axes
                  'plot_points' : None} # dict of points to plot

box_params = {'llcrnrlat' : 68.99,
              'urcrnrlat' : 69.31,
              'llcrnrlon' : -49.8,
              'urcrnrlon' : -48.3,
              'color' : 'r'}

params = {'llcrnrlon' : -50.8, # first x-coord
          'llcrnrlat' : 68.32, # first y-coord
          'urcrnrlon' : 42, # second x-coord
          'urcrnrlat' : 70.1, # second y-coord
          'scale_color' : 'r',
          'scale_length' : 200,
          'scale_loc' : 2,
          'figsize' : (6,4),
          'lat_interval' : 0.5,
          'lon_interval' : 1.0,
          'plot_grid' : True,
          'plot_scale' : True,
          'axes_color' : 'k'}

cont_plot_params = {'width' : 0.8,
                    'height' : 1.2,
                    'loc' : 1}

close_params = {'llcrnrlat' : 68.99,
                'urcrnrlat' : 69.31,
                'llcrnrlon' : -49.8,
                'urcrnrlon' : -48.3,
                'scale_color' : bc,
                'scale_length' : 50,
                'scale_loc' : 1,
                'figsize' : (6,4),
                'lat_interval' : 0.05,
                'lon_interval' : 0.25,
                'plot_grid' : False,
                'plot_scale' : False,
                'axes_color' : 'r'}

Bmax = srffmodel.B.vector().max()
Bmin = srffmodel.B.vector().min()

Smax = srffmodel.S.vector().max()
Smin = srffmodel.S.vector().min()

Uobmax = srffmodel.U_ob.vector().max()
Uobmin = srffmodel.U_ob.vector().min()

B_lvls = np.array([-1250, -1000, -750, -500, -250, 0.0, 250, 500])
B_lvls_2 = np.array([-1300, -1200, -1150, -1100, -1050, -950, -900,
                    -850, -800, -700, -650, -600, -550, -450, -400,
                    -350, -300, -200, -150, -100, -50, 50, 100, 150,
                    200, 300, 350, 400, 450])
S_lvls = np.array([250, 500, 750, 1000, 1250, 1500])
S_lvls_2 = np.array([150, 200, 300, 350, 400, 450, 550, 600,
                    650, 700, 800, 850, 900, 950, 1050, 1100,
                    1150, 1200, 1300, 1350, 1400])
U_ob_lvls = np.array([Uobmin, 50, 100, 250, 500, 1000, 2500, Uobmax])

# plot :
#=====

plotIce(drg, srffmodel.B, name='B', direc=out_dir,
        title='', cmap='RdGy', scale='lin',
        levels=B_lvls, levels_2=B_lvls_2, tp=True, tpAlpha=0.3,
        contour_type='lines', cb=False,
        extend='neither', show=False, ext='.pdf',
        zoom_box=False, zoom_box_kwargs=zoom_box_kwargs,
        params=close_params, plot_pts=plot_pts)

plotIce(drg, srffmodel.S, name='S', direc=out_dir,
        title='', cmap='gist_yarg', scale='lin',
        levels=S_lvls, levels_2=S_lvls_2, tp=True, tpAlpha=0.3,
        contour_type='lines', cb=False,
        extend='neither', show=False, ext='.pdf',
        zoom_box=False, zoom_box_kwargs=zoom_box_kwargs,
        params=close_params, plot_pts=plot_pts)

plotIce(drg, srffmodel.U_ob, name='region', direc=out_dir,
        title=r'$\mathbf{u}_{ob}$', cmap=cmap, scale='lin',
        levels=U_ob_lvls, tp=True, tpAlpha=0.4, box_params=box_params,
        extend='neither', show=False, ext='.pdf', cb_format="%i",
        params=params, plot_continent=True, cont_plot_params=cont_plot_params)

```

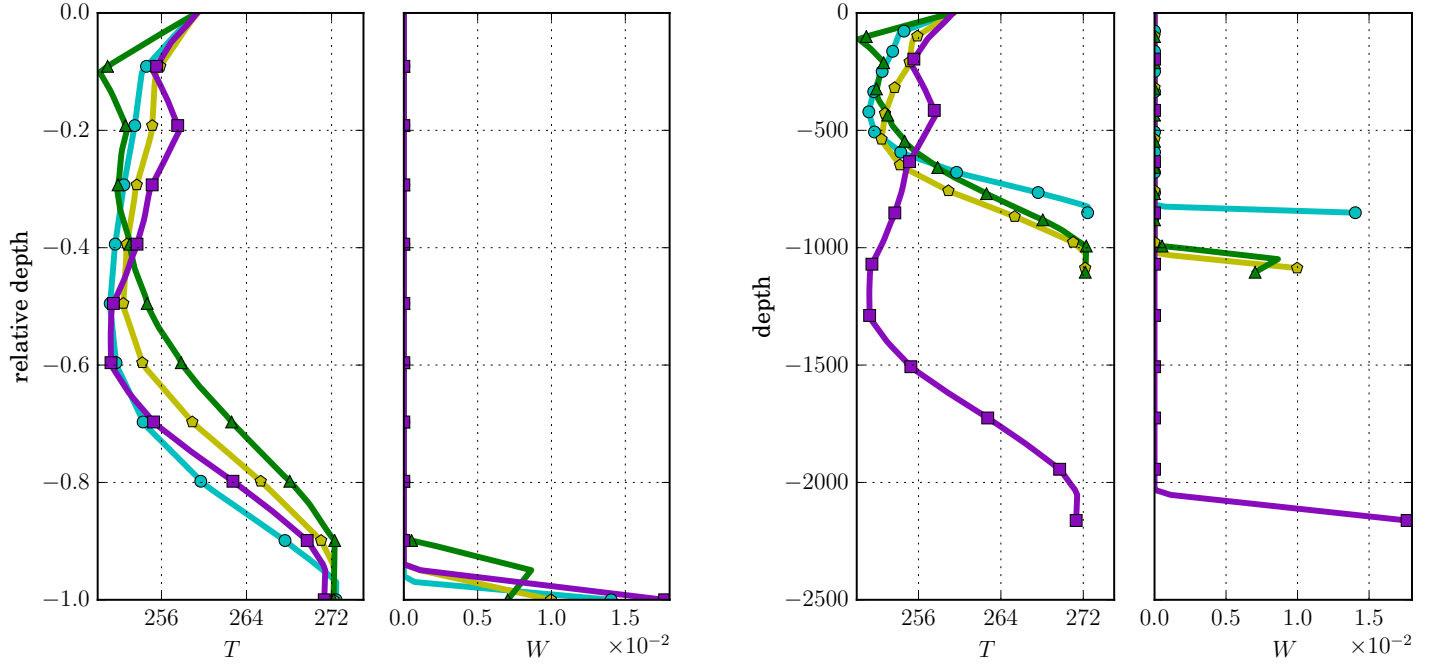



Figure 16.3: Temperature T in degrees K and unit-less water content W profiles with relative-depth-vertical coordinates (left) and actual depth coordinates (right). The colored points correspond to the latitude/longitude coordinates in Figure 16.2, and the points indicate vertex locations of the mesh from which the data were interpolated.

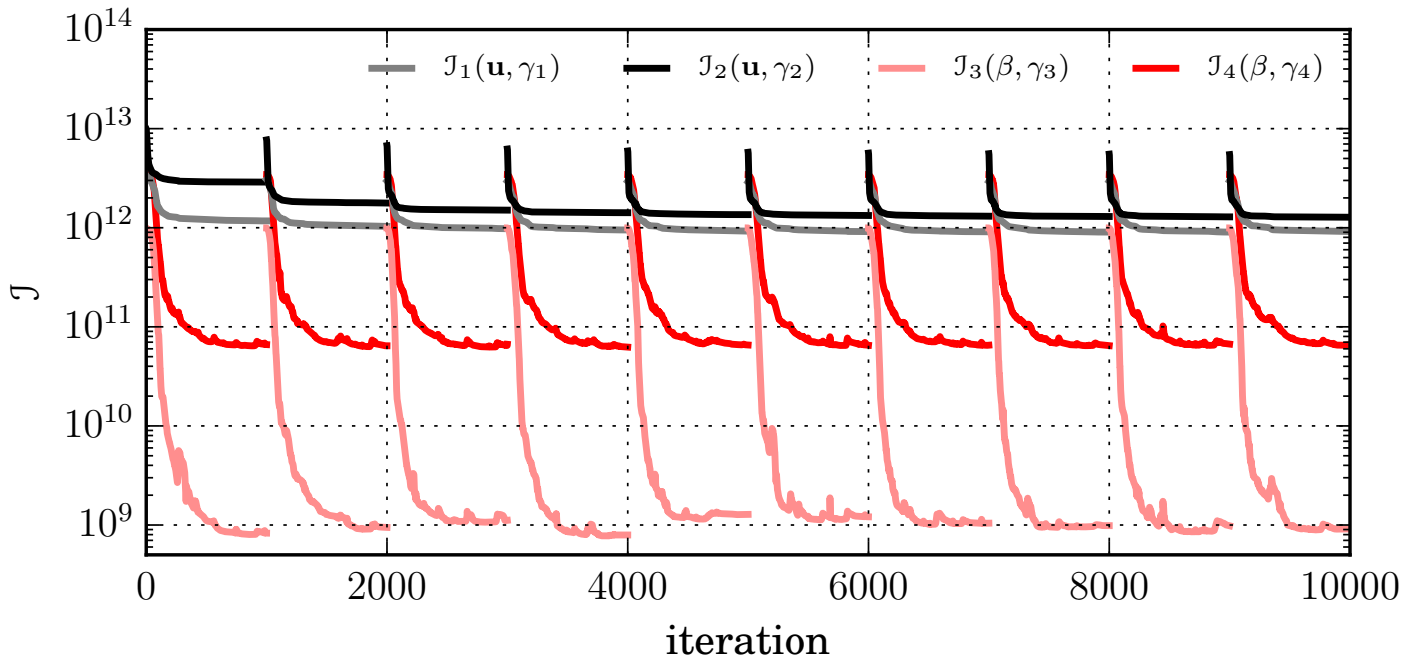


Figure 16.4: Convergence plots of momentum objective functional (12.1) for Algorithm 6 using maximum basal water content $W_c = 0.01$. The individual components include logarithmic cost functional (black), L^2 cost functional (dark gray), Tikhonov regularization functional (light red), and total variation regularization functional (red). The peaks located every 1000 iterations indicate iterations of Algorithm 6.

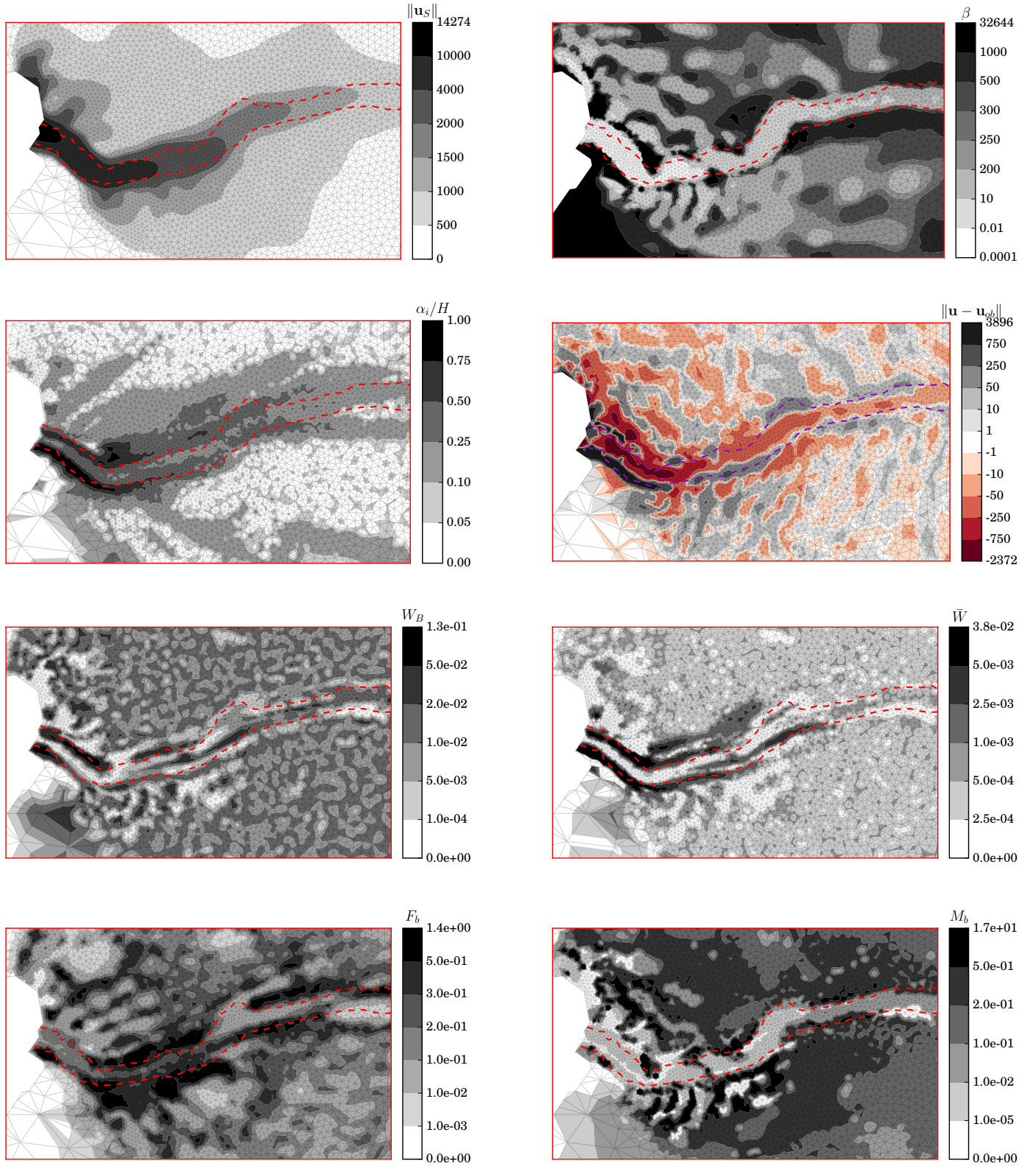


Figure 16.5: Results of TMC-inversion procedure 6 with maximum basal water content $W_c = 0.01$ for region of Western Greenland's Jakobshavn Glacier. The optimized surface velocity magnitude $\|\mathbf{u}_S\|$ (top left), optimized basal traction β (top right), ratio of ice column that is temperate (top middle left), velocity magnitude misfit (top middle right), basal water content W (bottom middle left), vertically averaged water content (bottom middle right), optimized basal water discharge F_b (bottom left) and basal melt rate M_b (bottom right). The dashed lines indicate the -500 m depth contour of basal topography B depicted in Figure 16.2.

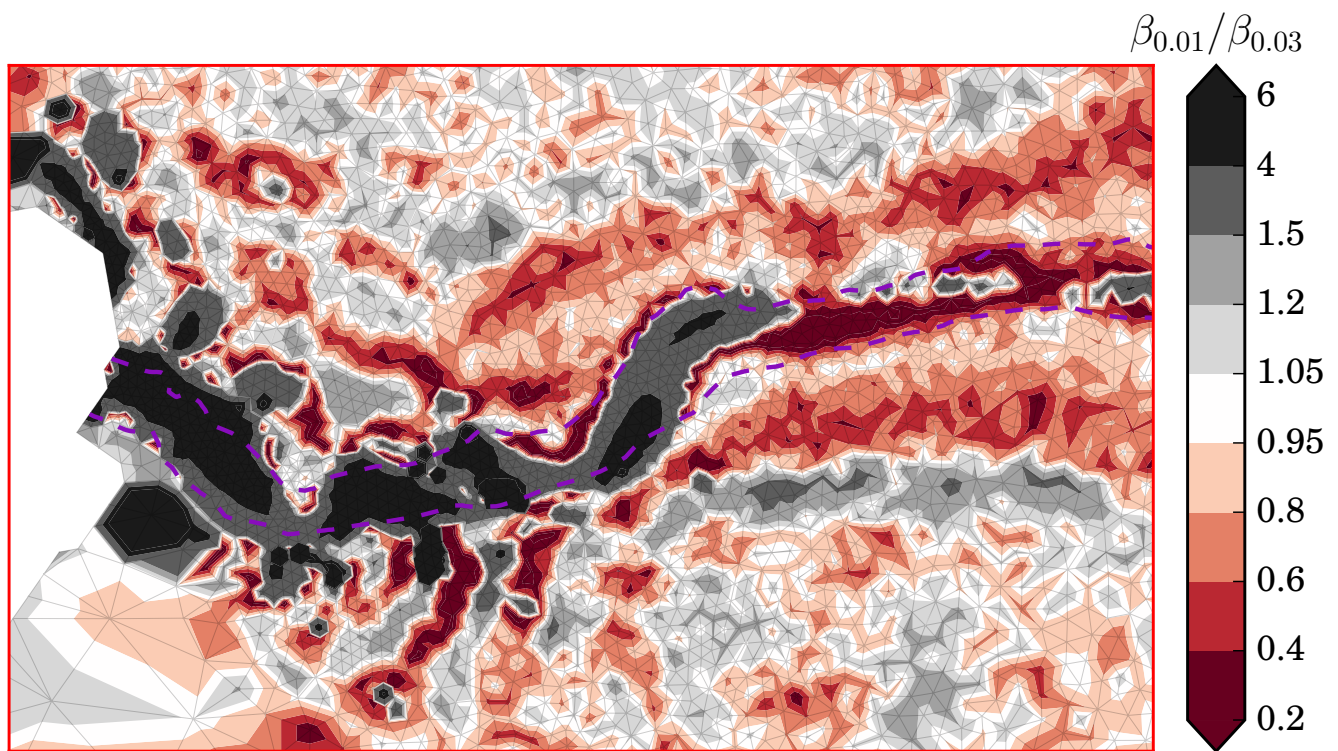
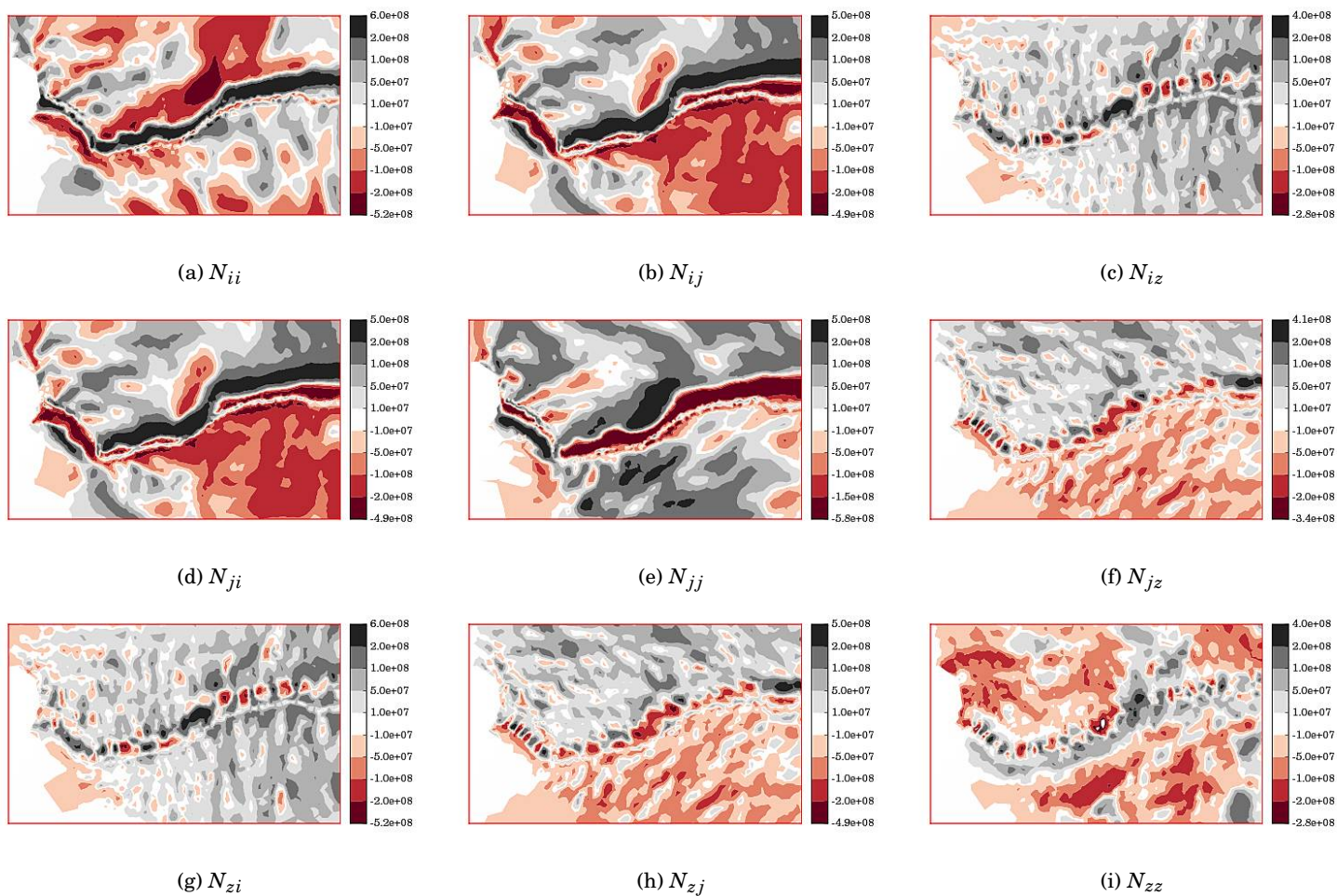
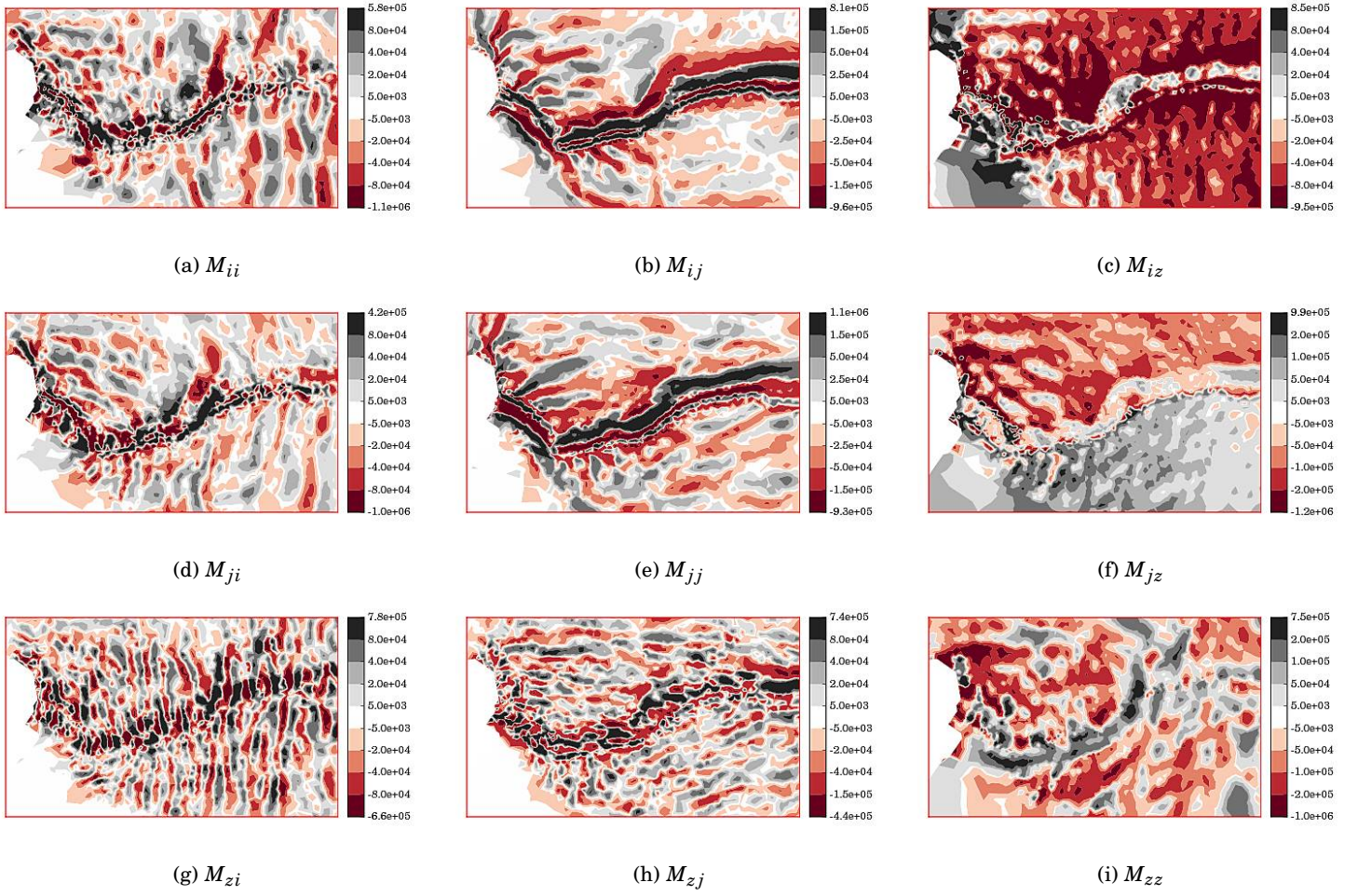


Figure 16.6: Ratio of basal traction attained by Algorithm 6 between using maximum basal water content $W_c = 0.01$ and $W_c = 0.03$. These results were obtained using fewer momentum optimization iterations, resulting in traction fields which are more irregular than that depicted in Figure 16.5.

Figure 16.7: Membrane stress N_{kk} .

Figure 16.8: Membrane stress balance M_{kk} .

Appendix A

Jump condition at the basal surface

In order to clearly illustrate the derivation of latent energy flux (10.34), the process described in § 9.3 of Greve and Blatter (2009) is rewritten using the differing notation and energy definition (10.14) used here.

First, the general jump condition of a singular surface σ located within the material volume Ω is

$$\forall \mathbf{x} \in \sigma : \quad \llbracket \psi \rrbracket(\mathbf{x}, t) = \psi^+(\mathbf{x}, t) - \psi^-(\mathbf{x}, t), \quad (\text{A1})$$

where $\mathbf{x} = [x \ y \ z]^\top$ is the position vector of the surface, t is time, and

$$\begin{aligned} \forall \mathbf{x} \in \sigma : \quad \psi^-(\mathbf{x}, t) &= \lim_{\mathbf{y} \rightarrow \mathbf{x}, \mathbf{y} \in \Omega^-} \psi(\mathbf{y}, t) \\ \psi^+(\mathbf{x}, t) &= \lim_{\mathbf{y} \rightarrow \mathbf{x}, \mathbf{y} \in \Omega^+} \psi(\mathbf{y}, t). \end{aligned}$$

By convention, the positive side of the ice base is identified with the lithosphere and negative side with the ice.

Next, the *mass jump condition* of a substance s with density ρ_s on singular surfaces is defied as

$$\llbracket \rho_s(\mathbf{u}_s - \mathbf{w}) \cdot \mathbf{n} \rrbracket = P, \quad (\text{A2})$$

where \mathbf{u}_s is the substance velocity, \mathbf{w} is the velocity of the singular surface, \mathbf{n} is the outward-pointing normal vector, and P is the rate of production of the substance on the singular surface. For the mixture here, we have some component of ice and water, denoted with subscripts i and w , respectively, with *barycentric velocity*

$$\mathbf{u} = \frac{1}{\rho} (\bar{\rho}_i \mathbf{u}_i + \bar{\rho}_w \mathbf{u}_w). \quad (\text{A3})$$

with partial densities $\bar{\rho}_i$ and $\bar{\rho}_w$ defined as the mass of ice and water per unit volume of the mixture. Thus, the water content of the mixture is defined as

$$W = \frac{\bar{\rho}_w}{\rho}, \quad (1 - W) = \frac{\bar{\rho}_i}{\rho}. \quad (\text{A4})$$

In addition, a non-advective water mass flux \mathbf{j} describes the water motion relative to the motion of barycentre (A3),

$$\mathbf{j} = \bar{\rho}_w(\mathbf{u}_w - \mathbf{u}) = \rho W(\mathbf{u}_w - \mathbf{u}). \quad (\text{A5})$$

Next, the *mass balance for the component water* is defined as

$$\frac{\partial \bar{\rho}_w}{\partial t} + \nabla \cdot (\bar{\rho}_w \mathbf{u}_w) = M,$$

where M is the rate of water mass produced per unit mixture volume. Applying water content definition (A4) and non-advective water mass flux (A5), this is equivalent to

$$\rho \dot{W} = -\nabla \cdot \mathbf{j} + M, \quad (\text{A6})$$

where the Newton-overdot notation ($\dot{\cdot}$) denotes time differentiation. The constitutive relations used to close the system are

$$\dot{\theta} = a \dot{T}_m + b T_m \dot{T}_m + L_f \dot{W} \quad (\text{A7})$$

$$\mathbf{j} = -\tilde{v} \nabla W \quad (\text{A8})$$

$$\mathbf{q}_s = -k(T_m) \nabla T_m, \quad (\text{A9})$$

where $\tilde{v} = v/L_f$ is the ‘water diffusivity’ as presented in Greve (1997), and energy definition (10.14) was used to derive energy time derivative (A7). The second relation is Fick’s diffusion law for the motion of water, and the last term is the sensible energy flux using Fourier’s law of heat conduction. Using this notation, the total heat flux is

$$\mathbf{q} = \mathbf{q}_s + \mathbf{q}_l = \mathbf{q}_s + L_f \mathbf{j},$$

and using the stress and strain constitutive relation expressed through shear viscosity (8.10) in strain-heat definition (8.11), the *mixture energy balance* is thus (Greve and Blatter, 2009)

$$\rho \dot{\theta} = -\nabla \cdot (\mathbf{q}_s + L_f \mathbf{j}) + Q. \quad (\text{A10})$$

Introducing constitutive relations (A7 – A9, 8.11) into water mass balance (A6) and mixture energy balance (A10) yield respectively

$$\rho \dot{W} = \tilde{v} \nabla \cdot \nabla W + M \quad (\text{A11})$$

and

$$\rho (a \dot{T}_m + b T_m \dot{T}_m + L_f \dot{W}) = \nabla \cdot (k \nabla T_m) + L_f \tilde{v} \nabla \cdot \nabla W + Q. \quad (\text{A12})$$

Solving for the water content time derivative term,

$$L_f \rho \dot{W} = \nabla \cdot (k \nabla T_m) + L_f \tilde{v} \nabla \cdot \nabla W + Q - \rho a \dot{T}_m - \rho b T_m \dot{T}_m,$$

and inserting (A11), the expression for the water production rate is therefore

$$L_f M = \nabla \cdot (k \nabla T_m) + Q - \rho a \dot{T}_m - \rho b T_m \dot{T}_m.$$

Next, the *mass jump relation for the component water* is defined using (A2),

$$\llbracket \tilde{\rho}_w(\mathbf{u}_w - \mathbf{w}) \cdot \mathbf{n} \rrbracket = \rho M_b,$$

where the water-production term $P = \rho M_b$ in (A2) has been defined using basal melting rate (10.36). Using general jump condition (A1), we have

$$\tilde{\rho}_w^- F_b - \tilde{\rho}_w^+(\mathbf{u}_w^+ - \mathbf{w}) \cdot \mathbf{n} = \rho M_b,$$

with *water mass flux into the base* $\tilde{\rho}_w^- F_b$ defined with basal water discharge

$$F_b = (\mathbf{u}_w^- - \mathbf{w}) \cdot \mathbf{n}.$$

Using water content definition (A4) and assuming the water content on the lithosphere side is composed entirely of water,

$$\rho W(\mathbf{u}_w - \mathbf{w}) \cdot \mathbf{n} = \rho_w F_b - \rho M_b, \quad (\text{A13})$$

Next, the *mass jump relation for the component ice* is similarly defined as

$$\llbracket \tilde{\rho}_i(\mathbf{u}_i - \mathbf{w}) \cdot \mathbf{n} \rrbracket = -\rho M_b.$$

Because the lithosphere is impermeable to ice, as evident by impenetrability condition (9.4), this simplifies to

$$\tilde{\rho}_i(\mathbf{u}_i - \mathbf{w}) \cdot \mathbf{n} = \rho(1 - W)(\mathbf{u}_i - \mathbf{w}) \cdot \mathbf{n} = \rho M_b. \quad (\text{A14})$$

Next, using barycentric velocity (A3) and water content (A4), it follows that

$$\mathbf{u} - \mathbf{w} = W(\mathbf{u}_w - \mathbf{w}) + (1 - W)(\mathbf{u}_i - \mathbf{w}),$$

which upon scalar multiplication by \mathbf{n} and use of water jump (A13) and ice jump (A14), we have

$$(\mathbf{u} - \mathbf{w}) \cdot \mathbf{n} = \frac{\rho_w}{\rho} F_b.$$

Using non-advective water mass flux (A5), component water jump (A13), and component ice jump (A14), we have the flux of water normal to the basal boundary

$$\begin{aligned} \mathbf{j} \cdot \mathbf{n} &= \rho W(\mathbf{u}_w - \mathbf{u}) \cdot \mathbf{n} \\ &= \rho W(\mathbf{u}_w - \mathbf{w}) \cdot \mathbf{n} - \rho W(\mathbf{u} - \mathbf{w}) \cdot \mathbf{n} \\ &= \rho_w F_b - \rho M_b - W \rho_w F_b \\ &= (1 - W) \rho_w F_b - \rho M_b \approx \rho_w F_b - \rho M_b. \end{aligned}$$

Finally, from water-flux constitutive relation (A8) we have

$$\begin{aligned} (\nu \nabla W) \cdot \mathbf{n} &= (L_f \tilde{\nu} \nabla W) \cdot \mathbf{n} = -L_f \mathbf{j} \cdot \mathbf{n} \\ &= \rho L_f M_b - (1 - W) \rho_w L_f F_b \\ &\approx \rho L_f M_b - \rho_w L_f F_b, \end{aligned}$$

Hence water flux boundary condition (10.34) has been derived.

□

Appendix B

Leibniz formula

Leibniz formula, referred to as *Leibniz's rule for differentiating an integral with respect to a parameter that appears in the integrand and in the limits of integration*, states that

$$\begin{aligned} \frac{d}{dx} \int_{a(x)}^{b(x)} F(x, y) dy = & \int_{a(x)}^{b(x)} F_x(x, y) dy \\ & + F(x, b(x))b'(x) - F(x, a(x))a'(x), \end{aligned}$$

where F and F_x are both continuous over the domain $[a, b]$.

Proof:

Let

$$I(x, a, b) = \frac{d}{dx} \int_{a(x)}^{b(x)} F(x, y) dy.$$

Then

$$\frac{dI}{dx} = \frac{\partial I}{\partial x} \frac{\partial x}{\partial x} + \frac{\partial I}{\partial a} \frac{\partial a}{\partial x} + \frac{\partial I}{\partial b} \frac{\partial b}{\partial x},$$

and

$$\frac{\partial I}{\partial x} = \frac{\partial}{\partial x} \int_a^b F(x, y) dy = \int_{a(x)}^{b(x)} F_x(x, y) dy$$

$$\frac{\partial x}{\partial x} = 1$$

$$\frac{\partial I}{\partial a} = \frac{\partial}{\partial a} \int_{a(x)}^{b(x)} F(x, y) dy = \frac{\partial}{\partial a} [f(x, b(x)) - f(x, a(x))] = -F(x, a(x))$$

$$\frac{\partial a}{\partial x} = a'(x)$$

$$\frac{\partial I}{\partial b} = \frac{\partial}{\partial b} \int_{a(x)}^{b(x)} F(x, y) dy = \frac{\partial}{\partial b} [f(x, b(x)) - f(x, a(x))] = F(x, b(x))$$

$$\frac{\partial b}{\partial x} = b'(x),$$

and thus

$$\frac{dI}{dx} = \int_{a(x)}^{b(x)} F_x(x, y) dy + F(x, b(x))b'(x) - F(x, a(x))a'(x) \quad \square$$

Bibliography

- Ahrens, James, Berk Geveci, and Charles Law (2005). *ParaView: An End-User Tool for Large Data Visualization*. Elsevier. ISBN: 978-0123875822.
- Arnold, D. N., F. Brezzi, and M. Fortin (1984). “A stable finite element for the Stokes equations”. In: *Calcolo* 21.4, pp. 337–344.
- Arthern, Robert J., Dale P. Winebrenner, and David G. Vaughan (2006). “Antarctic snow accumulation mapped using polarization of 4.3-cm wavelength microwave emission”. In: *Journal of Geophysical Research: Atmospheres* 111.D6. D06107, n/a–n/a. ISSN: 2156-2202. DOI: 10.1029/2004JD005667. URL: <http://dx.doi.org/10.1029/2004JD005667>.
- Aschwanden, A. and H. Blatter (2009). “Mathematical modeling and numerical simulation of polythermal glaciers”. In: *Journal of Geophysical Research: Earth Surface* 114.F1. F01027, n/a–n/a. ISSN: 2156-2202. DOI: 10.1029/2008JF001028. URL: <http://dx.doi.org/10.1029/2008JF001028>.
- Aschwanden, A. et al. (2012). “An enthalpy formulation for glaciers and ice sheets”. In: *Journal of Glaciology* 58.209, pp. 441–457.
- Bamber, J. L. et al. (2013). “A new bed elevation dataset for Greenland”. In: *The Cryosphere* 7.2, pp. 499–510. DOI: 10.5194/tc-7-499-2013. URL: <http://www.the-cryosphere.net/7/499/2013/>.
- Blatter, H. (1995). “Velocity and stress fields in grounded glaciers: a simple algorithm for including deviatoric stress gradients”. In: *Journal of Glaciology* 41.138, pp. 333–344.
- Blatter, H. and R. Greve (2015). “Comparison and verification of enthalpy schemes for polythermal glaciers and ice sheets with a one-dimensional model”. In: *Polar Science* 9.2, pp. 196–207. DOI: arXiv:1410.6251v4[physics.ao-ph].
- Brinkerhoff, D. J. and J. V. Johnson (2013). “Data assimilation and prognostic whole ice sheet modelling with the variationally derived, higher order, open source, and fully parallel ice sheet model VarGlaS”. In: *The Cryosphere* 7.4, pp. 1161–1184. DOI: 10.5194/tc-7-1161-2013. URL: <http://www.the-cryosphere.net/7/1161/2013/>.
- (2015). “A stabilized finite element method for calculating balance velocities in ice sheets”. In: *Geoscientific Model Development* 8.5, pp. 1275–1283. DOI: 10.5194/gmd-8-1275-2015. URL: <http://www.geosci-model-dev.net/8/1275/2015/>.
- Brooks, A.N. and T.J.R. Hughes (1982). “Streamline upwind/Petrov-Galerkin formulation for convection dominated flows with particular emphasis on the incompressible Navier-Stokes equations”. In: *Computer Methods in Applied Mechanics and Engineering* 32.1-3, pp. 199–259. ISSN: 0045-7825. DOI: [http://dx.doi.org/10.1016/0045-7825\(82\)90071-8](http://dx.doi.org/10.1016/0045-7825(82)90071-8). URL: <http://www.sciencedirect.com/science/article/pii/0045782582900718>.
- Bryson, Arthur E. and Yu-Chi Ho (1975). *Applied Optimal Control*. Halsted Press.
- Burgess, Evan W. et al. (2010). “A spatially calibrated model of annual accumulation rate on the Greenland Ice Sheet (1958–2007)”. In: *Journal of Geophysical Research F: Earth Surface* 115. URL: <http://www.scopus.com/inward/record.url?eid=2-s2.0-77951082793&partnerID=40&md5=a6430b7a7a93a85254e4f44589f3b0f4>.
- Byrd, R.H. et al. (1995). “A limited memory algorithm for bound constrained optimization”. In: *SIAM Journal on Scientific and Statistical Computing* 16.5, pp. 1190–1208.
- Codina, R. (1998). “Comparison of some finite element methods for solving the diffusion-convection-reaction equation”. In: *Computer Methods in Applied Mechanics and Engineering* 156.1-4, pp. 185–210.
- Codina, R., E. Oñate, and M. Cervera (1992). “The intrinsic time for the streamline upwind/Petrov-Galerkin formulation using quadratic elements”. In: *Computer methods in applied mechanics and engineering* 94, pp. 239–262.
- Davis, John M (2013). *Introduction to Applied Partial Differential Equations*. 1st edition. New York, New York: W.H. Freeman and Company.
- Douglas Jr, Jim and Junping Wang (1989). “An absolutely stabilized finite element method for the Stokes problem”. In: *Mathematics of computation* 52.186, pp. 495–508.
- Dukowicz, J.K. (2012). “Reformulating the full-Stokes ice sheet model for a more efficient computational solution”. In: *The Cryosphere* 6.1, pp. 21–34. DOI: 10.5194/tc-6-21-2012. URL: <http://www.the-cryosphere.net/6/21/2012/>.
- Dukowicz, J.K., S.F. Price, and W.H. Lipscomb (Aug. 2010). “Consistent approximations and boundary conditions for ice-sheet dynamics from a principle of least action”. In: *Journal of Glaciology* 56, pp. 480–496. DOI: 10.3189/002214310792447851.
- (2011). “Incorporating arbitrary basal topography in the variational formulation of ice-sheet models”. In: *Journal of Glaciology* 57, pp. 461–467. DOI: 10.3189/002214311796905550.
- Elman, Howard, David Silvester, and Andy Wathen (2005). *Finite Elements and Fast Iterative Solvers*. Oxford University Press.

- Farrell, P. E. et al. (2013). "Automated Derivation of the Adjoint of High-Level Transient Finite Element Programs". In: *SIAM Journal on Scientific Computing* 35.4, pp. C369–C393. DOI: 10.1137/120873558. eprint: <http://dx.doi.org/10.1137/120873558>. URL: <http://dx.doi.org/10.1137/120873558>.
- Fausto, Robert S. et al. (2009). "A new present-day temperature parameterization for Greenland". In: *Journal of Glaciology* 55.189, pp. 95–105.
- Fowler, A. C. (1982). "On the transport of moisture in polythermal glaciers". In: *Geophysical & Astrophysical Fluid Dynamics* 28 (2), pp. 99–140. DOI: 10.1080/03091928408222846.
- Fretwell, P. et al. (2013). "Bedmap2: improved ice bed, surface and thickness datasets for Antarctica". In: *The Cryosphere* 7.1, pp. 375–393. DOI: 10.5194/tc-7-375-2013. URL: <http://www.the-cryosphere.net/7/375/2013/>.
- Freund, Jouni and Rolf Stenberg (1995). "On weakly imposed boundary conditions for second order problems". In: *pre-seedings of the International Conference on Finite Elements in Fluids – New trends and applications, Venezia*.
- Geuzaine, Christophe and Jean-François Remacle (2009). "Gmsh: A 3-D finite element mesh generator with built-in pre- and post-processing facilities". In: *International Journal for Numerical Methods in Engineering* 79.11, pp. 1309–1331. ISSN: 1097-0207. DOI: 10.1002/nme.2579. URL: <http://dx.doi.org/10.1002/nme.2579>.
- Glen, J. W. (1952). "Experiments on the Deformation of Ice". In: *Journal of Glaciology* 2.12, pp. 111–114. ISSN: 0022-1430.
- Greve, R. (1997). "A continuum-mechanical formulation for shallow polythermal ice sheets". In: *Philosophical Transactions of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 355.1726, pp. 921–974. ISSN: 1364-503X. DOI: 10.1098/rsta.1997.0050.
- Greve, R. and H. Blatter (2009). *Dynamics of Ice Sheets and Glaciers*. Springer.
- Greve, R., T. Zwinger, and Y. Gong (2014). "On the pressure dependence of the rate factor in Glen's flow law". In: *Journal of Glaciology* 60, pp. 397–398. DOI: 10.3189/2014JoG14J019.
- Hansen, Per Christian (1992). "Analysis of Discrete Ill-Posed Problems by Means of the L-Curve". In: *SIAM Review* 34.4, pp. 561–580. DOI: 10.1137/1034115. eprint: <http://dx.doi.org/10.1137/1034115>. URL: <http://dx.doi.org/10.1137/1034115>.
- Hill, R. (1950). *The Mathematical Theory of Plasticity*. Oxford University Press.
- Hughes, T.J.R. (1995). "Multiscale phenomena: Green's functions, the Dirichlet-to-Neumann formulation, subgrid scale models, bubbles and the origins of stabilized methods". In: *Computer Methods in Applied Mechanics and Engineering* 127.1-4, pp. 387–401. ISSN: 0045-7825. DOI: [http://dx.doi.org/10.1016/0045-7825\(95\)00844-9](http://dx.doi.org/10.1016/0045-7825(95)00844-9). URL: <http://www.sciencedirect.com/science/article/pii/S0045782595008449>.
- Hughes, T.J.R. and L.P. Franca (1987). "A new finite element formulation for computational fluid dynamics: VII. The Stokes problem with various well-posed boundary conditions: symmetric formulations that converge for all velocity/pressure spaces". In: *Computer methods in applied mechanics and engineering* 65, pp. 85–96.
- Hughes, T.J.R., L.P. Franca, and M. Balestra (1986). "A new finite element formulation for computational fluid dynamics: V. Circumventing the Babuska-Brezzi condition: A stable Petrov-Galerkin formulation of the Stokes problem accommodating equal-order interpolation". In: *Computer methods in applied mechanics and engineering* 59, pp. 85–99.
- Hughes, T.J.R., L.P. Franca, and G.M. Hulbert (1989). "A new finite element formulation for computational fluid dynamics: VIII. The Galerkin/least-squares method for advective-diffusive equations". In: *Computer methods in applied mechanics and engineering* 73, pp. 173–189.
- Hunter, J. D. (2007). "Matplotlib: A 2D graphics environment". In: *Computing In Science & Engineering* 9.3, pp. 90–95. DOI: 10.5281/zenodo.44579.
- Hutter, Kolumban (1982). "A mathematical model of polythermal glaciers and ice sheets". In: *Geophysical & Astrophysical Fluid Dynamics* 21.3-4, pp. 201–224. DOI: 10.1080/03091928208209013. eprint: <http://dx.doi.org/10.1080/03091928208209013>. URL: <http://dx.doi.org/10.1080/03091928208209013>.
- Johnson, Claes (2009). *Numerical Solution of Partial Differential Equations by the Finite Element Method*. Dover Publications.
- Kleiner, T. et al. (2015). "Enthalpy benchmark experiments for numerical ice sheet models". In: *The Cryosphere* 9.1, pp. 217–228. DOI: 10.5194/tc-9-217-2015. URL: <http://www.the-cryosphere.net/9/217/2015/>.
- Le Brocq, A. M., A. J. Payne, and A. Vieli (2010). "An improved Antarctic dataset for high resolution numerical ice sheet models (ALBMAP v1)". In: *Earth System Science Data* 2.2, pp. 247–260. DOI: 10.5194/essd-2-247-2010. URL: <http://www.earth-syst-sci-data.net/2/247/2010/>.
- Lliboutry, L. (1996). "Temperate ice permeability, stability of water veins and percolation of internal meltwater". In: *Journal of Glaciology* 42.141, pp. 201–211. DOI: doi:10.3198/1996JoG42-141-201-211. URL: <http://www.ingentaconnect.com/content/igsoc/jog/1996/00000042/00000141/art00002>.
- Logan, John David (2006). *Applied Mathematics*. 3rd edition. Hoboken, New Jersey: John Wiley and Sons.
- Logg, Anders, Kent-Andre Mardal, and Garth Wells (2012). *Automated Solution of Differential Equations by the Finite Element Method*. Springer Berlin Heidelberg.
- Lüthi, Martin et al. (2002). "Mechanisms of fast flow in Jakobshavn Isbrae, West Greenland: Part III. Measurements of ice deformation, temperature and cross-borehole conductivity in boreholes to the bedrock". In: *Journal of Glaciology* 48.162, pp. 369–385. DOI: doi:10.3189/172756502781831322. URL: <http://www.>

- ingentaconnect . com / content / igsoc / jog / 2002 / 00000048/00000162/art00003.
- MacAyeal, Douglas R. (1993). "A tutorial on the use of control methods in ice-sheet modeling". In: *Journal of Glaciology* 39.131.
- Morlighem, M., H. Seroussi, and E. Rignot (2013). "Inversion of basal friction in Antarctica using exact and incomplete adjoints of a higher-order model". In: *Journal of Geophysical Research: Earth Surface* 118, pp. 1746–1753. DOI: 10.1002/jgrf.20125.
- Nitsche, J. (1970/71). "Über ein Variationsprinzip zur Lösung von Dirichlet-Problemen bei Verwendung von Teilräumen, die keinen Randbedingungen unterworfen sind". In: *Abhandlungen aus dem Mathematischen Seminar der Universität Hamburg* 36, pp. 9–15.
- Nocedal, Jorge and Stephen Wright (2000). *Numerical Optimization*. Springer.
- Nye, J. F. (1957). "The Distribution of Stress and Velocity in Glaciers and Ice-Sheets". In: *Proceedings of the Royal Society of London A: Mathematical, Physical and Engineering Sciences* 239.1216, pp. 113–133. ISSN: 0080-4630. DOI: 10.1098/rspa.1957.0026. eprint: <http://rspa.royalsocietypublishing.org/content/239/1216/113.full.pdf>. URL: <http://rspa.royalsocietypublishing.org/content/239/1216/113>.
- Nye, J. F. and F. C. Frank (1973). "Hydrology of the intergranular veins in a temperate glacier". In: *Symposium on the Hydrology of Glaciers* 95, pp. 157–161.
- Paterson, W. S. B. (1994). *Physics of Glaciers*. 3rd edition. Butterworth-Heinemann.
- Patterson, W.S.B. and W.F. Budd (1982). "Flow parameters for ice sheet modelling". In: *Cold Regions Science and Technology* 6.2, pp. 175–177. ISSN: 0165-232X. DOI: [http://dx.doi.org/10.1016/0165-232X\(82\)90010-6](http://dx.doi.org/10.1016/0165-232X(82)90010-6). URL: <http://www.sciencedirect.com/science/article/pii/0165232X82900106>.
- Pattyn, F. (2003). "A new three-dimensional higher-order thermomechanical ice sheet model: Basic sensitivity, ice stream development, and ice flow across subglacial lakes". In: *Journal of Geophysical Research* 108.B8. DOI: 10.1029/2002JB002329.
- Pattyn, F. et al. (2008). "Benchmark experiments for higher-order and full-Stokes ice sheet models (ISMIPa€HOM)". In: *The Cryosphere* 2.2, pp. 95–108. DOI: 10.5194/tc-2-95-2008. URL: <http://www.the-cryosphere.net/2/95/2008/>.
- Petra, Noemi et al. (2012). "An inexact Gauss–Newton method for inversion of basal sliding and rheology parameters in a nonlinear Stokes ice sheet model". In: *Journal of Glaciology* 58.211, pp. 889–903. ISSN: 0022-1430. DOI: doi:10.3189/2012JoG11J182.
- Pham, Q. T. (1995). "Comparison of general-purpose finite-element methods for the Stefan problem". In: *Numerical Heat Transfer, Part B: Fundamentals* 27.4, pp. 417–435. DOI: 10.1080/10407799508914965. eprint: <http://dx.doi.org/10.1080/10407799508914965>. URL: <http://dx.doi.org/10.1080/10407799508914965>.
- Pérez, Fernando and Brian E. Granger (2007). "IPython: A System for Interactive Scientific Computing". In: *Computing in Science & Engineering* 9.3, pp. 21–29. DOI: <http://dx.doi.org/10.1109/MCSE.2007.53>. URL: <http://scitation.aip.org/content/aip/journal/cise/9/3/10.1109/MCSE.2007.53>.
- Raymond, C. F. and W. D. Harrison (1975). "Some Observations on the Behavior of the Liquid and Gas Phases in Temperate Glacier Ice". In: *Journal of Glaciology* 14.71, pp. 213–233. DOI: doi:10.3198/1975jog14-71-213-233. URL: <http://www.ingentaconnect.com/content/igsoc/jog/1975/00000014/00000071/art00001>.
- Reddy, John N (1993). *An Introduction to the Finite Element Method*. 2nd edition. McGraw-Hill.
- Rignot, E. and J. Mouginot (2012). "Ice flow in Greenland for the International Polar Year 2008–2009". In: *Geophysical Research Letters* 39.11. L11501, n/a–n/a. ISSN: 1944-8007. DOI: 10.1029/2012GL051634. URL: <http://dx.doi.org/10.1029/2012GL051634>.
- Rignot, E., J. Mouginot, and B. Scheuchl (2011). *MEaSUREs InSAR-Based Antarctica Ice Velocity Map*. Boulder, Colorado USA: NASA DAAC at the National Snow and Ice Data Center. DOI: 10.5067/MEASURES/CRYOSPHERE/nsidc-0484.001.
- Ritz, Catherine (1987). "Time dependent boundary conditions for calculation of temperature fields in ice sheets". In: *The Physical Basis of Ice Sheet Modelling* 170, pp. 207–216.
- Shreve, R. L. (1972). "Movement of Water in Glaciers*". In: *Journal of Glaciology* 11.62, pp. 205–214. DOI: doi:10.3198/1972JoG11-62-205-214. URL: <http://www.ingentaconnect.com/content/igsoc/jog/1972/00000011/00000062/art00002>.
- Taylor, C. and P. Hood (1973). "A numerical solution of the Navier-Stokes equations using the finite element technique". In: *Computers & Fluids* 1.1, pp. 73–100. ISSN: 0045-7930. DOI: [http://dx.doi.org/10.1016/0045-7930\(73\)90027-3](http://dx.doi.org/10.1016/0045-7930(73)90027-3). URL: <http://www.sciencedirect.com/science/article/pii/0045793073900273>.
- Veen, C. J. van der et al. (2001). "Trend surface analysis of Greenland accumulation". In: *Journal of Geophysical Research: Atmospheres* 106.D24, pp. 33909–33918. ISSN: 2156-2202. DOI: 10.1029/2001JD900156. URL: <http://dx.doi.org/10.1029/2001JD900156>.
- Vogel, C. (2002). *Computational Methods for Inverse Problems*. Society for Industrial and Applied Mathematics. DOI: 10.1137/1.9780898717570. eprint: <http://epubs.siam.org/doi/pdf/10.1137/1.9780898717570>. URL: <http://epubs.siam.org/doi/abs/10.1137/1.9780898717570>.
- Wächter, Andreas and Lorenz T. Biegler (2006). "On the implementation of an interior-point filter line-search algorithm for large-scale nonlinear programming". English. In: *Mathematical Programming* 106.1, pp. 25–57. ISSN: 0025-5610. DOI: 10.1007/s10107-004-0559-y. URL: <http://dx.doi.org/10.1007/s10107-004-0559-y>.

- Walt, Stéfan van der, S. Chris Colbert, and Gaël Varoquaux (2011). "The NumPy Array: A Structure for Efficient Numerical Computation". In: *Computing in Science & Engineering* 13.2, pp. 22–30. DOI: <http://dx.doi.org/10.1109/MCSE.2011.37>. URL: <http://scitation.aip.org/content/aip/journal/cise/13/2/10.1109/MCSE.2011.37>.
- Watkins, David S. (2010). *Fundamentals of Matrix Computations*. John Wiley and Sons.
- Yen, Yin-Chao (1981). "Review of thermal properties of snow, ice and sea ice". In: *U.S. Army Cold Regions Research and Engineering Laboratory Report No. 81-10*.

Index

- Adjoint method, *see also* Control theory
- Adjoint operator, 31
- Armijo condition, 44
- Automatic differentiation, 42

- Backtracking line search, 45
- Balance equations
 - Basal energy, 74
 - Internal energy, 72
 - Mass, 55
 - Momentum, 55
 - Stress, 125
 - Velocity, 103
- Basal freeze-on, 75
- Basal melting rate, 75
- Basal traction, 91
- Basal water discharge, 74
- BFGS Algorithm, 44
- Bilinear form, 5
- Boundary conditions
 - Essential, 3, 7
 - Natural, 3, 7
- Boundary layer, 11, 35
- Bubble functions, 32

- Chain rule, 135
- Clausius-Clapeyron relationship, 72
- Cold/temperate surface, 71
- Constitutive ice-flow relation, 51
- Constrained optimization
 - Adjoint variable, 47, 79, 91
 - Control parameter, 78, 91
 - Cost function, 91
 - Forward model, 79
 - Interior point methods, 47
 - L-curve analysis, 94
 - Lagrangian, 47, 79, 91
 - Necessary conditions, 48
 - Objective function, 47, 78, 91
 - State parameter, 47, 78, 91
 - Tikhonov regularization function, 91
 - Total-variation regularization function, 91
- Control theory, 47
- Cryostatic stress, 58, 60

- Data assimilation, 91
- Differential operator, 31
- Diffusivity, 73

- Directional derivative, 14, 42, 125
- Divergence Theorem, 103
- Driving stress, 104

- Eigenvalue problem, 15
- Enthalpy, 73
- Enthalpy-gradient method, 73

- FEniCS, 8
- Finite-element interpolation, 5
- Flow enhancement factor, 74, 96
- Flow-rate factor, 74
- Fourier series method, 15
- Functionals, 47

- Gâteaux derivative, 42
- Galerkin method, 5
- Global element matrix, 7
- Green's functions, 32

- Heat capacity
 - Latent, 72
 - Sensible, 72
- Hydrostatic stress, 55

- Ice age, 135
- Impenetrability, 55, 58, 62
- Inner product, 5
- Internal energy, 72
- Interpolation properties, 5
- Intrinsic-time parameter
 - Age equation, 135
 - Balance velocity, 105
 - General form, 34
 - Internal energy, 76
- ISMIP-HOM simulations, 63, 96, 127

- Kronecker delta, 5

- L^2 space, 5
- Lagrange interpolation functions, *see also* Shape functions
- Lagrange multiplier, 47
- Latent energy flux, 74
- Leibniz's Rule, 104, 151
- Linear differential equations
 - 1D, 11, 12, 14, 34, 36
 - 2D, 19–21, 37, 63, 106
 - 3D, 27, 28, 63, 96, 127
- Local element matrix, 6

- Log-barrier method, 48, 79, 91
- Mass matrix, 16
- Material derivative, 135
- Membrane stress, 125
- Mixed methods, 21, 22
- Newton-Raphson method, 41, 48
- Nitsche method, 21, 37
- Non-linear differential equations
 - 1D, 43
 - 2D, 85
 - 3D, 137
- Numerical integration, 13
- Péclet number
 - General form, 34
 - Internal energy, 76
- Plane-strain simulations, 63, 85
- Poisson equation, 19, 27
- Projection, 14
- Quasi-Newton methods, 44
- Residual, 41, 79
- Reynold's Transport Theorem, 103
- Secant equation, 44
- Shape functions, 5
- Shelf inversion, *see also* Flow enhancement factor
- Singular perturbation, 11
- Sobolev space, 5
- Stabilization methods
 - Galerkin/least-squares, 34
 - Streamline-upwind/Petrov-Galerkin, 34
 - Subgrid-scale-model, 34
- Static condensation, 33
- Stiffness matrix, 6
- Stokes equations
 - Applied to ice, first-order, 57
 - Applied to ice, full-Stokes, 55
 - Applied to ice, plane-strain, 59
 - Applied to ice, reformulated-Stokes, 60
 - Applied to ice, shallow ice approximation, 93
 - No-slip, 20, 28
 - Slip-friction, 21, 37, 55, 57, 59, 60
 - Stability, 21
- Strain heat, 51
- Subgrid scales, 31
- Temperate zone, 71
- Tensor
 - Cauchy-stress, 51
 - Deviatoric stress, 51
 - Effective strain-rate, 51
 - Effective stress, 51
 - First-order strain-rate, 126
 - Membrane stress, 126
 - Plane-strain strain rate, 59
 - Plane-strain stress, 59
 - Reformulated-Stokes strain-rate, 60
 - Reformulated-Stokes stress, 60
 - Strain-rate, 51
- Test functions, 5
- Thermal conductivity, 72
- Thermo-mechanical coupling, 83
- Transient problem, 15
- Trial functions, 5
- Variational form, 4
- Variational principle
 - Euler-Lagrange equations arising from, 62
 - First-order, 58
 - Full-Stokes, 55
 - Plane-strain, 59
 - Reformulated-Stokes, 61
 - Variational forms, 62
- Viscosity, 51
- Water content of ice, 72
- Weak form, 5